

---

# AWS OpsWorks

## User Guide

### API Version 2013-02-18



## AWS OpsWorks: User Guide

Copyright © 2014 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

The following are trademarks of Amazon Web Services, Inc.: Amazon, Amazon Web Services Design, AWS, Amazon CloudFront, Cloudfront, Amazon DevPay, DynamoDB, ElastiCache, Amazon EC2, Amazon Elastic Compute Cloud, Amazon Glacier, Kindle, Kindle Fire, AWS Marketplace Design, Mechanical Turk, Amazon Redshift, Amazon Route 53, Amazon S3, Amazon VPC. In addition, Amazon.com graphics, logos, page headers, button icons, scripts, and service names are trademarks, or trade dress of Amazon in the U.S. and/or other countries. Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon.

All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

## Table of Contents

What is AWS OpsWorks? .....	1
Stacks .....	2
Layers .....	3
Instances .....	3
Apps .....	4
Customizing your Stack .....	4
Resource Management .....	5
Security and Permissions .....	5
Monitoring and Logging .....	5
CLI, SDK, and AWS CloudFormation Templates .....	6
Where to Start .....	6
Documentation Roadmap .....	7
Getting Started .....	9
Step 1: Sign in to the AWS OpsWorks Console .....	10
Step 2: Create a Simple Application Server Stack .....	10
Step 1: Create a Stack .....	11
Step 2: Add a PHP App Server Layer .....	13
Step 3: Add an Instance to the PHP App Server Layer .....	14
Step 4: Create and Deploy an App .....	16
Step 3: Add a Back-end Data Store .....	20
Step 1: Add a Back-end Database .....	20
Step 2: Update SimplePHPApp .....	21
A Short Digression: Cookbooks, Recipes, and AWS OpsWorks Attributes .....	23
Step 3: Add the Custom Cookbooks to MyStack .....	28
Step 4: Run the Recipes .....	28
Step 5: Deploy SimplePHPApp, Version 2 .....	29
Step 6: Run SimplePHPApp .....	30
Step 4: Scale Out MyStack .....	31
Step 1: Add a Load Balancer .....	32
Step 2: Add PHP App Server Instances .....	35
Step 3: Monitor MyStack .....	35
Delete MyStack .....	37
Stacks .....	40
Create a New Stack .....	41
Running a Stack in a VPC .....	44
VPC Basics .....	44
Create a VPC for an AWS OpsWorks Stack .....	46
Create a Stack in the VPC .....	48
Update a Stack .....	50
Clone a Stack .....	51
Run Stack Commands .....	52
Use Custom JSON to Modify the Stack Configuration JSON .....	53
Shut Down a Stack .....	55
Layers .....	57
OpsWorks Layer Basics .....	58
How to Create an OpsWorks Layer .....	58
How to Edit an OpsWorks Layer .....	59
How to Use Auto Healing to Replace a Layer's Failed Instances .....	67
How to Delete an OpsWorks Layer .....	68
Load Balancer Layers .....	69
Elastic Load Balancing .....	70
HAProxy AWS OpsWorks Layer .....	72
Database Layers .....	77
Amazon RDS Service Layer .....	77
MySQL OpsWorks Layer .....	80

Application Server Layers .....	81
Java App Server AWS OpsWorks Layer .....	82
Node.js App Server AWS OpsWorks Layer .....	88
PHP App Server AWS OpsWorks Layer .....	91
Rails App Server AWS OpsWorks Layer .....	92
Static Web Server AWS OpsWorks Layer .....	95
Custom AWS OpsWorks Layers .....	96
Other Layers .....	98
Ganglia Layer .....	98
Memcached .....	100
Instances .....	101
Operating Systems .....	101
Amazon Linux .....	102
Ubuntu LTS .....	103
Adding an Instance to a Layer .....	103
Using Custom AMIs .....	107
Manually Starting, Stopping, and Rebooting 24/7 Instances .....	109
Changing Instance Properties .....	111
Deleting Instances .....	112
Managing Load with Time-based and Load-based Instances .....	113
How Automatic Time-based Scaling Works .....	114
How Automatic Load-based Scaling Works .....	116
How Load-based Scaling Differs from Auto Healing .....	119
Using SSH to Communicate with an Instance .....	119
Using the MindTerm SSH Client .....	120
Using a Third-Party Client with Amazon EC2 Key Pairs .....	122
Apps .....	125
Adding Apps .....	125
Configuring an App .....	126
Deploying Apps .....	132
Other Deployment Commands .....	134
Editing Apps .....	134
Connecting an Application to a Database Server .....	135
Java App Server .....	136
Node.js App Server .....	137
PHP App Server .....	137
Rails App Server .....	138
Using a Custom Recipe .....	139
Using Environment Variables .....	141
Java App Server .....	142
Node.js App Server .....	142
PHP App Server .....	142
Rails App Server .....	142
Custom App Servers .....	143
Passing Data to Applications .....	143
Using Repository SSH Keys .....	146
Using Custom Domains .....	146
Running Multiple Applications on the Same Application Server .....	147
Using SSL .....	148
Step 1: Install and Configure OpenSSL .....	148
Step 2: Create a Private Key .....	149
Step 3: Create a Certificate Signing Request .....	150
Step 4: Submit the CSR to Certificate Authority .....	150
Step 5: Edit the App .....	151
Cookbooks and Recipes .....	153
Cookbook Repositories .....	154
Cookbook Components .....	155
Attributes .....	156

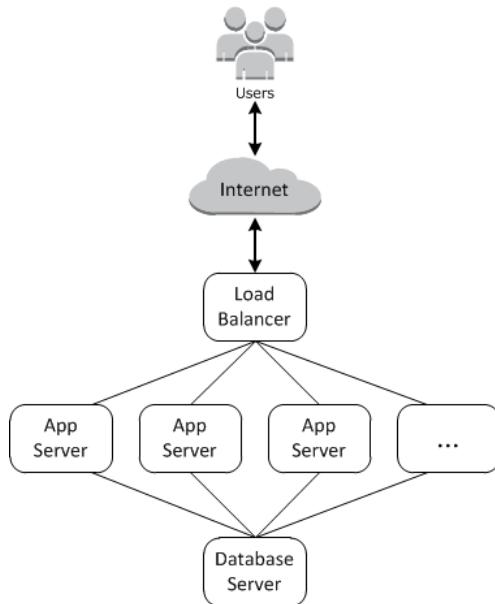
Templates .....	158
Recipes .....	159
Chef Versions .....	161
Implementing Recipes for Chef 11.4 Stacks .....	162
Implementing Recipes for Chef 11.10 Stacks .....	163
Migrating an Existing Stack to a new Chef Version .....	169
Ruby Versions .....	170
Installing Custom Cookbooks .....	171
Specifying a Custom Cookbook Repository .....	172
Updating Custom Cookbooks .....	174
Executing Recipes .....	175
AWS OpsWorks Lifecycle Events .....	176
Automatically Running Recipes .....	177
Manually Running Recipes .....	178
Cookbooks 101 .....	179
Vagrant and Test Kitchen .....	180
Cookbook Basics .....	181
Implementing Cookbooks for AWS OpsWorks .....	209
Resource Management .....	218
Registering Resources with a Stack .....	219
Registering Amazon EBS Volumes with a Stack .....	219
Registering Elastic IP Addresses with a Stack .....	220
Registering Amazon RDS Instances with a Stack .....	221
Attaching and Moving Resources .....	222
Assigning Amazon EBS Volumes to an Instance .....	223
Associating Elastic IP Addresses with an Instance .....	224
Attaching Amazon RDS Instances to an App .....	226
Detaching Resources .....	226
Unassigning Amazon EBS Volumes .....	226
Disassociating Elastic IP Addresses .....	227
Detaching Amazon RDS Instances .....	227
Deregistering Resources .....	228
Deregistering Amazon EBS Volumes .....	228
Deregistering Elastic IP Addresses .....	228
Deregistering Amazon RDS Instances .....	229
Customizing AWS OpsWorks .....	230
Customizing AWS OpsWorks Configuration by Overriding Attributes .....	231
Attribute Precedence .....	232
Overriding Attributes Using Custom JSON .....	233
Overriding AWS OpsWorks Attributes Using Custom Cookbook Attributes .....	235
Extending AWS OpsWorks Configuration Files Using Custom Templates .....	236
Extending a Built-in Layer .....	237
Using Recipes to Run Scripts .....	237
Using Chef Deployment Hooks .....	238
Running Cron Jobs .....	239
Installing and Configuring Packages .....	240
Creating a Custom Tomcat Server Layer .....	240
Attributes File .....	241
Setup Recipes .....	242
Configure Recipes .....	250
Deploy Recipes .....	254
Create a Stack and Run an Application .....	256
Stack Configuration and Deployment JSON .....	262
Configure JSON .....	262
Deploy JSON .....	265
Using Stack Configuration JSON Values in Custom Recipes .....	267
How to Obtain a Stack Configuration JSON .....	267
Monitoring .....	268

Monitoring Stacks using Amazon CloudWatch .....	268
Metrics .....	269
Stack Metrics .....	270
Layer Metrics .....	270
Instance Metrics .....	271
Logging AWS OpsWorks API Calls By Using AWS CloudTrail .....	272
AWS OpsWorks Information in CloudTrail .....	273
Understanding AWS OpsWorks Log File Entries .....	273
Using Amazon CloudWatch Logs with AWS OpsWorks .....	275
Security and Permissions .....	276
Managing User Permissions .....	277
Managing Users .....	278
Managing Users' Permissions Using a Stack's Permissions Page .....	282
Managing Permissions by Attaching an IAM policy .....	284
Example Policies .....	286
AWS OpsWorks Permissions Levels .....	290
Signing in as an IAM User .....	291
Setting an IAM User's Public SSH Key .....	292
Allowing AWS OpsWorks to Act on Your Behalf .....	293
Specifying Permissions for Apps Running on EC2 instances .....	296
Using AWS OpsWorks with Other AWS Services .....	299
Using Other Back-end Data Stores .....	300
How to Set up a Database Connection .....	300
How to Connect an Application Server Instance to Amazon RDS .....	301
Using ElastiCache Redis as an In-Memory Key-Value Store .....	305
Step 1: Create an ElastiCache Redis Cluster .....	305
Step 2: Set up a Rails Stack .....	307
Step 3: Create and Deploy a Custom Cookbook .....	307
Step 4: Assign the Recipe to a LifeCycle Event .....	310
Step 5: Add Access Information to the Stack Configuration JSON .....	311
Step 6: Deploy and run the App .....	312
Using an Amazon S3 Bucket .....	313
Step 1: Create an Amazon S3 Bucket .....	313
Step 2: Create a PHP App Server Stack .....	315
Step 3: Create and Deploy a Custom Cookbook .....	316
Step 4: Assign the Recipes to LifeCycle Events .....	318
Step 5: Add Access Information to the Stack Configuration JSON .....	319
Step 6: Deploy and Run PhotoApp .....	320
Using the AWS OpsWorks CLI .....	323
Create an Instance (create-instance) .....	324
Create an Instance with a Themed Host Name .....	324
Deploy an App (create-deployment) .....	325
Migrate the Rails Database .....	326
Describe a Stack's Apps (describe-apps) .....	326
Describe a Stack's Commands (describe-commands) .....	327
Describe a Stack's Deployments (describe-deployments) .....	328
Describe a Stack's Elastic IP Addresses (describe-elastic-ips) .....	329
Describe a Stack's Instances (describe-instances) .....	329
Describe Stacks (describe-stacks) .....	330
Describe a Stack's Layers (describe-layers) .....	331
Execute a Recipe (create-deployment) .....	334
Install Dependencies (create-deployment) .....	335
Update the Stack Configuration (update-stack) .....	335
Debugging and Troubleshooting Guide .....	337
Debugging Recipes .....	337
Chef Logs .....	338
Using the Agent CLI .....	344
Running Cookbook Tests .....	347

Common Debugging and Troubleshooting Issues .....	347
Debugging Custom Recipes .....	348
Troubleshooting AWS OpsWorks .....	349
Appendix A: AWS OpsWorks Layer Reference .....	354
HAProxy Layer Reference .....	354
MySQL Layer Reference .....	356
Web Server Layers Reference .....	357
Java App Server Layer Reference .....	357
Node.js App Server Layer Reference .....	358
PHP App Server Layer Reference .....	360
Rails App Server Layer Reference .....	361
Static Web Server Layer Reference .....	362
Custom Layer Reference .....	363
Other Layers .....	364
Ganglia Layer Reference .....	364
Memcached Layer Reference .....	366
Appendix B: Instance Agent CLI .....	368
agent_report .....	369
get_json .....	369
instance_report .....	372
list_commands .....	373
run_command .....	373
show_log .....	374
stack_state .....	375
Appendix C: AWS OpsWorks Attribute Reference .....	378
Stack Configuration and Deployment JSON Attributes .....	379
opsworks Attributes .....	380
opsworks_custom_cookbooks Attributes .....	392
dependencies Attributes .....	392
ganglia Attributes .....	392
mysql Attributes .....	393
passenger Attributes .....	393
opsworks_bundler Attributes .....	394
deploy Attributes .....	394
Other Top-Level Attributes .....	399
Built-in Recipe Attributes .....	400
apache2 Attributes .....	400
deploy Attributes .....	406
haproxy Attributes .....	407
memached Attributes .....	410
mysql Attributes .....	411
nginx Attributes .....	415
opsworks_java Attributes .....	418
passenger_apache2 Attributes .....	421
ruby Attributes .....	423
unicorn Attributes .....	424
Resources .....	427
Reference Guides, Tools, and Support Resources .....	427
AWS Software Development Kits .....	428
History .....	429

# What is AWS OpsWorks?

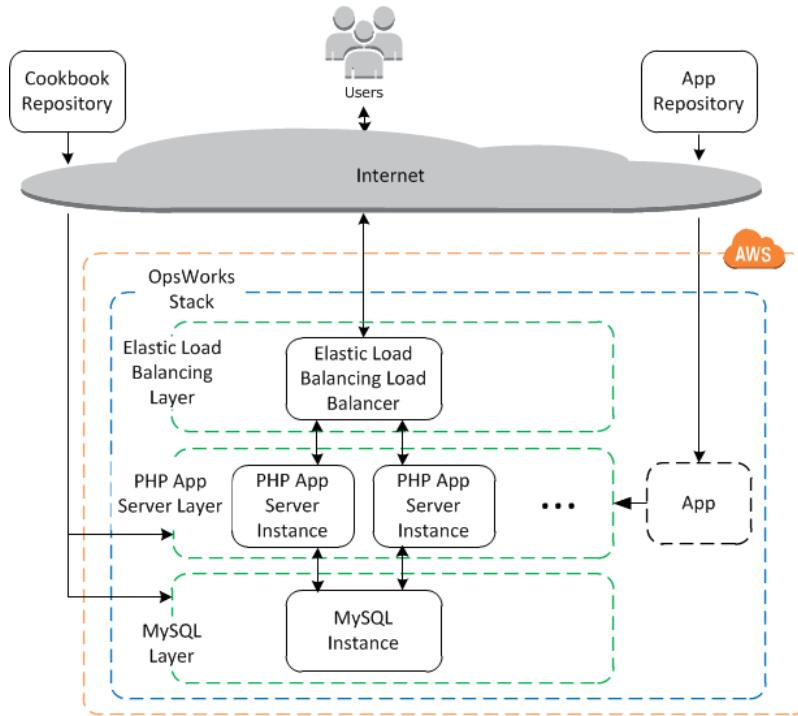
Cloud-based computing usually involves groups of AWS resources, such as EC2 instances, EBS volumes, and so on, which must be created and managed collectively. For example, a web application typically requires application servers, database servers, load balancers, and so on. This group of instances is typically called a *stack*; a simple application server stack might look something like the following.



In addition to creating the instances and installing the necessary packages, you typically need a way to distribute applications to the application servers, monitor the stack's performance, manage security and permissions, and so on.

AWS OpsWorks provides a simple and flexible way to create and manage stacks and applications. It supports a standard set of components—including application servers, database servers, load balancers, and more—that you can use to assemble your stack. These components all come with a standard configuration and are ready to run. However, you aren't limited to the standard components and configurations. AWS OpsWorks gives you the tools to customize the standard package configurations, install additional packages, and even create your own custom components. AWS OpsWorks also provides a way to manage related AWS resources, such as Elastic IP addresses and Amazon EBS volumes.

Here's what a basic PHP application server stack might look with AWS OpsWorks. It consists of a group of PHP application servers running behind an Elastic Load Balancing load balancer, with a backend MySQL database server.



Although relatively simple, this stack shows all the key AWS OpsWorks features. Here's how it's put together.

### Topics

- [Stacks \(p. 2\)](#)
- [Layers \(p. 3\)](#)
- [Instances \(p. 3\)](#)
- [Apps \(p. 4\)](#)
- [Customizing your Stack \(p. 4\)](#)
- [Resource Management \(p. 5\)](#)
- [Security and Permissions \(p. 5\)](#)
- [Monitoring and Logging \(p. 5\)](#)
- [CLI, SDK, and AWS CloudFormation Templates \(p. 6\)](#)
- [Where to Start \(p. 6\)](#)

## Stacks

The **stack** is the core AWS OpsWorks component. It is basically a container for AWS resources—Amazon EC2 instances, Amazon EBS volumes, Elastic IP addresses, and so on—that have a common purpose and would be logically managed together. The stack helps you manage these resources as a group and also defines some default configuration settings, such as the instances' operating system and AWS region. If you want to isolate some stack components from direct user interaction, you can run the stack in a VPC.

## Layers

You define the stack's components by adding one or more *layers*. A layer is basically a blueprint that specifies how to configure a set of Amazon EC2 instances for a particular purpose, such as serving applications or hosting a database server. You assign each instance to at least one layer, which determines what packages are to be installed on the instance, how they are configured, whether the instance has an Elastic IP address or Amazon EBS volume, and so on.

AWS OpsWorks includes a set of built-in layers that support the following standard scenarios:

- Application server: Java App Server, Node.js App Server, PHP App Server, Rails App Server, Static Web Server
- Database server: Amazon RDS and MySQL
- Load balancer: Elastic Load Balancing, HAProxy
- Monitoring server: Ganglia
- In-memory key-value store: Memcached

If the built-in layers don't quite meet your requirements, you can customize or extend them by modifying packages' default configurations, adding custom Chef recipes to perform tasks such as installing additional packages, and more. You can also customize layers to work with AWS services that are not natively supported, such as using Amazon RDS as a database server. If that's still not enough, you can create a fully custom layer, which gives you complete control over which packages are installed, how they are configured, how applications are deployed, and more.

## Instances

Each layer has at least one *instance*. An instance represents an Amazon EC2 instance and defines its basic configuration, such as operating system and size. Other configuration settings, such as Elastic IP addresses or Amazon EBS volumes, are defined by the instance's layer. In addition, each layer has an associated set of [Chef recipes](#) that AWS OpsWorks runs on the layer's instances at key points in an instance's life cycle. A layer's Setup recipes perform tasks such as installing and configuring the layer's packages and starting daemons; the Deploy recipes handle downloading applications; and so on. You can also run recipes manually, at any time.

When you start an instance, AWS OpsWorks launches an Amazon EC2 instance using the configuration settings specified by the instance and its layer. After the Amazon EC2 has booted, AWS OpsWorks installs an agent that handles communication between the instance and the service. AWS OpsWorks then runs the layer's Setup recipes to install, configure, and start the layer's software, followed by the Deploy recipes, which install any applications.

For example, after you start an instance that belongs to the example's PHP App Server layer, AWS OpsWorks launches an Amazon EC2 instance and then runs a set of recipes that install, configure, and start a PHP application server and install PHP applications. An instance can even belong to multiple layers. In that case, AWS OpsWorks runs the recipes for each layer so you can, for example, have an instance that supports a PHP application server and a MySQL database server.

You can start instances in several ways.

- **24/7 instances** are started manually and run until you stop them.
- **Load-based instances** are automatically started and stopped by AWS OpsWorks, based on specified load metrics, such as CPU utilization. They allow your stack to automatically adjust the number of instances to accommodate variations in incoming traffic.

- **Time-based instances** are run by AWS OpsWorks on a specified daily and weekly schedule. They allow your stack to automatically adjust the number of instances to accommodate predictable usage patterns.

AWS OpsWorks also supports autoheal instances. If an agent stops communicating with the service, AWS OpsWorks automatically stops and restarts the instance.

## Apps

You store applications and related files in a repository such as an Amazon S3 bucket. Each application is represented by an **app**, which specifies the application type and contains the information that AWS OpsWorks needs to deploy the application from the repository to your instances. You can deploy apps in two ways:

- Automatically—When you start an app server instance, AWS OpsWorks automatically deploys all apps of the appropriate type; Java apps are deployed to the Java App Server layer's instances, and so on.
- Manually—if you have a new app or want to update an existing one, you can manually deploy it to your online instances.

When you deploy an app, AWS OpsWorks runs the Deploy recipes on the stack's instances. The app server layer's Deploy recipes download the app from the repository to the instance and perform related tasks such as configuring the server and restarting the daemon. You typically have AWS OpsWorks run the Deploy recipes on the entire stack, which allows the other layers' instances to modify their configuration appropriately. However, you can limit deployment to a subset of instances if, for example, you want to test a new app before deploying it to every app server instance.

## Customizing your Stack

The built-in layers provide standard functionality that is sufficient for many purposes. However, you might customize them a bit. AWS OpsWorks provides a variety of ways to customize layers to meet your specific requirements:

- You can modify how AWS OpsWorks configures packages by overriding attributes that represent the various configuration settings, or by even overriding the templates used to create configuration files.
- You can extend an existing layer by providing your own custom recipes to perform tasks such as running scripts or installing and configuring nonstandard packages.
- You can create a fully custom layer by providing a set of recipes that handle all the tasks of installing packages, deploying apps, and so on.

You package your custom recipes and related files in one or more *cookbooks* and store the cookbooks in a repository such as Amazon S3. You can run your custom recipes manually, but AWS OpsWorks also lets you automate the process by supporting a set of five *lifecycle events*:

- **Setup** occurs on a new instance after it successfully boots.
- **Configure** occurs on all of the stack's instances when an instance enters or leaves the online state.
- **Deploy** occurs when you deploy an app.
- **Undeploy** occurs when you delete an app.
- **Shutdown** occurs when you stop an instance.

Each layer comes with a set of built-in recipes assigned to each of these events. When a lifecycle event occurs on a layer's instance, AWS OpsWorks runs the associated recipes. For example, when a Deploy event occurs on an app server instance, AWS OpsWorks runs the layer's built-in Deploy recipes to download the app.

You can run custom recipes automatically by assigning them to a layer's lifecycle events. AWS OpsWorks then automatically runs them for you after the built-in recipes have finished. For example, if you want to install an additional package on a layer's instances, write a recipe to install the package and assign it to the layer's Setup event. AWS OpsWorks will run it automatically on each of the layer's new instances, after the built-in Setup recipes are finished.

## Resource Management

With AWS OpsWorks, you can use any of your account's [Elastic IP address](#) and [Amazon EBS volume](#) resources in a stack. You can use the AWS OpsWorks console or API to register resources with a stack, attach registered resources to or detach them from instances, and move resources from one instance to another.

## Security and Permissions

AWS OpsWorks integrates with AWS Identity and Access Management (IAM) to provide robust ways of controlling how users access AWS OpsWorks, including the following:

- How individual users can interact with each stack, such as whether they can create stack resources such as layers and instances, or whether they can use SSH to connect to a stack's EC2 instances.
- How AWS OpsWorks can act on your behalf to interact with AWS resources such as Amazon EC2 instances.
- How apps that run on AWS OpsWorks instances can access AWS resources such as Amazon S3 buckets.
- How to manage public SSH keys and connect to an instance with SSH.

## Monitoring and Logging

AWS OpsWorks provides several features to help you monitor your stack and troubleshoot issues with your stack and any custom recipes:

- CloudWatch monitoring, which is summarized for your convenience on the OpsWorks **Monitoring** page.
- CloudTrail support, which logs API calls made by or on behalf of AWS OpsWorks in your AWS account.
- A Ganglia master layer that can be used to collect and display detailed monitoring data for the instances in your stack.
- An event log that lists all events in your stack.
- Chef logs that describe the details of what transpired for each lifecycle event on each instance, such as which recipes were run and what errors occurred.

# CLI, SDK, and AWS CloudFormation Templates

In addition to the graphical console, AWS OpsWorks also supports a command-line interface (CLI) and SDKs for multiple languages that can be used to perform any operation. Consider these features:

- The AWS OpsWorks CLI is part of the [AWS CLI](#), and can be used to perform any operation from the command-line.  
The AWS CLI supports multiple AWS services and can be installed on Windows, Linux, or OS X systems.
- AWS OpsWorks is included in [AWS Tools for Windows PowerShell](#) and can be used to perform any operation from a PowerShell command line.
- The AWS OpsWorks SDKs are included in the AWS SDK and can be used to perform any operation from applications implemented in: [Java](#), [JavaScript](#) (browser-based and Node.js), [.NET](#), [PHP](#), [Python \(boto\)](#), or [Ruby](#).

You can also use AWS CloudFormation templates to provision stacks. For some examples, see [AWS OpsWorks Snippets](#).

## Where to Start

The following video walks you through a basic example of how to use AWS OpsWorks: [Getting Started with AWS OpsWorks](#)

For a more detailed introduction to AWS OpsWorks that covers a broader range of features than the video, see [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#).

For a brief overview of the AWS OpsWorks documentation, see [Documentation Roadmap \(p. 7\)](#).

# Documentation Roadmap

---

This section provides a brief overview of the AWS OpsWorks documentation.

## [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#)

An introduction to AWS OpsWorks—directed primarily at new users—that walks you through the process of using AWS OpsWorks to set up a simple application server stack.

## [Stacks \(p. 40\)](#)

The stack is the top-level AWS OpsWorks entity. It represents a set of instances that you want to manage collectively, typically because they have a common purpose such as serving PHP applications. This chapter describes how to work with stacks, including how to create and clone stacks, and run stacks in a VPC.

## [Layers \(p. 57\)](#)

A layer is essentially a blueprint for an Amazon EC2 instance. It defines which packages and applications are installed, how they are configured, and so on. This chapter describes how to work with layers, including how to create and configure layers and a description of each built-in layer.

## [Instances \(p. 101\)](#)

Instances represent the EC2 instances that handle the work of serving applications, balancing traffic, and so on. This chapter describes how to work with instances, including how to create instances, start and stop instances, and connect to instances by using SSH.

## [Apps \(p. 125\)](#)

An app represents code that you want to run on an application server. This chapter describes how to work with apps, including how to create, update, and deploy them.

## [Cookbooks and Recipes \(p. 153\)](#)

AWS OpsWorks includes a built-in set of OpsCode Chef cookbooks that handle tasks such as installing and configuring packages and deploying apps for each layer. Many approaches to customizing a layer involve overriding or extending the built-in cookbooks by implementing one or more custom cookbooks. This chapter describes how to work with cookbooks, including a brief description of how to implement a custom cookbook, how to install custom cookbooks on your stack, and how to execute your custom recipes.

## [Resource Management \(p. 218\)](#)

You can use the **Resources** page to import your account's Elastic IP address and Amazon EBS volume resources into a stack and attach them to instances. This chapter describes how to use the **Resources** page to manage a stack's resources.

## [Customizing AWS OpsWorks \(p. 230\)](#)

If the built-in layers don't meet your specific requirements, you can customize them in a variety of ways. This chapter describes how to customize a stack, ranging from modifying individual configuration settings to creating fully custom layers.

[Monitoring \(p. 268\)](#)

AWS OpsWorks uses Amazon CloudWatch to provide metrics for a stack and summarizes them for your convenience on the **Monitoring** page. AWS OpsWorks is also integrated with CloudTrail, a service that captures API calls made by or on behalf of AWS OpsWorks in your AWS account. This chapter describes how to monitor your stack.

[Security and Permissions \(p. 276\)](#)

AWS OpsWorks integrates with AWS Identity and Access Management. This chapter describes how to manage security and permissions, including how users interact with AWS OpsWorks, how AWS OpsWorks and your applications interact with dependent AWS resources, and how to manage users' SSH keys.

[Using AWS OpsWorks with Other AWS Services \(p. 299\)](#)

A stack can use AWS services that are not directly integrated with AWS OpsWorks. This chapter describes how to use several AWS services, including Amazon S3, ElastiCache, and Amazon RDS.

[Using the AWS OpsWorks CLI \(p. 323\)](#)

The AWS OpsWorks command line interface (CLI) provides the same functionality as the console. This chapter describes how to use the CLI to perform a variety of common operations.

[Debugging and Troubleshooting Guide \(p. 337\)](#)

This chapter describes how to troubleshoot a variety of issues with stacks and custom recipes.

[Appendix A: AWS OpsWorks Layer Reference \(p. 354\)](#)

This appendix provides a detailed description of each built-in layer.

[Appendix B: Instance Agent CLI \(p. 368\)](#)

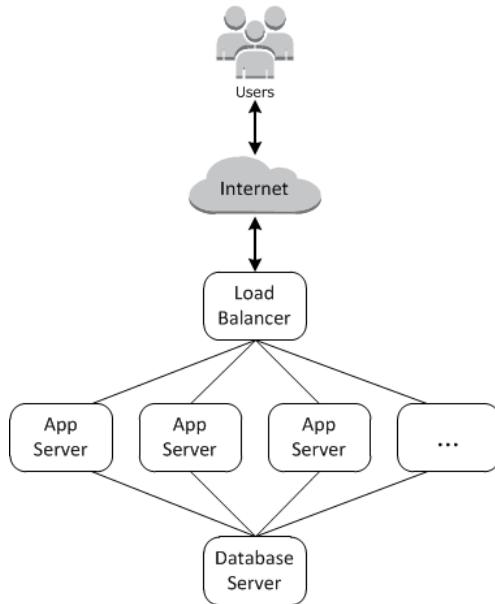
The agent that AWS OpsWorks installs on every instance exposes a command line interface (CLI). Using the CLI, you can perform a variety of tasks such as viewing logs by connecting to the instance with SSH and running the commands locally. This appendix describes how to use the agent CLI commands.

[Appendix C: AWS OpsWorks Attribute Reference \(p. 378\)](#)

AWS OpsWorks exposes the stack configuration settings and many of the package configuration settings as Chef attributes. You can use these attributes in custom recipes and override the default settings by using custom attributes files or by including custom attributes in a stack's configuration and deployment JSON. This appendix describes the commonly used attributes.

# Getting Started: Create a Simple PHP Application Server Stack

Cloud-based applications usually require a group of related resources—application servers, database servers, and so on—that must be created and managed collectively. This collection of instances is called a *stack*. A simple application stack might look something like the following.



The basic architecture consists of the following:

- A load balancer to distribute incoming traffic from users evenly across the application servers.
- A set of application server instances, as many as needed to handle the traffic.
- A database server to provide the application servers with a back-end data store.

In addition, you typically need a way to distribute applications to the application servers, monitor the stack, and so on.

AWS OpsWorks provides a simple and straightforward way to create and manage stacks and their associated applications and resources. This chapter introduces the basics of AWS OpsWorks—along with some of its more sophisticated features—by walking you through the process of creating the application server stack in the diagram. It uses an incremental development model that AWS OpsWorks makes easy to follow: Set up a basic stack and, once it's working correctly, add components until you arrive at a full-featured implementation.

- [Step 1: Sign in to the AWS OpsWorks Console \(p. 10\)](#) shows how to sign into the AWS OpsWorks console
- [Step 2: Create a Simple Application Server Stack \(p. 10\)](#) shows how to create a minimal stack that consists of a single application server.
- [Step 3: Add a Back-end Data Store \(p. 20\)](#) shows how to add a database server and connect it to the application server.
- [Step 4: Scale Out MyStack \(p. 31\)](#) shows how to scale out a stack to handle increased load by adding more application servers, and a load balancer to distribute incoming traffic.

#### Topics

- [Step 1: Sign in to the AWS OpsWorks Console \(p. 10\)](#)
- [Step 2: Create a Simple Application Server Stack \(p. 10\)](#)
- [Step 3: Add a Back-end Data Store \(p. 20\)](#)
- [Step 4: Scale Out MyStack \(p. 31\)](#)
- [Delete MyStack \(p. 37\)](#)

## Step 1: Sign in to the AWS OpsWorks Console

Before you can sign in, you must have an AWS account. If you already have one, you are automatically signed up for AWS OpsWorks. If you are new to AWS, or you would like to set up another account, follow these steps to open a new account:

#### To sign up for AWS

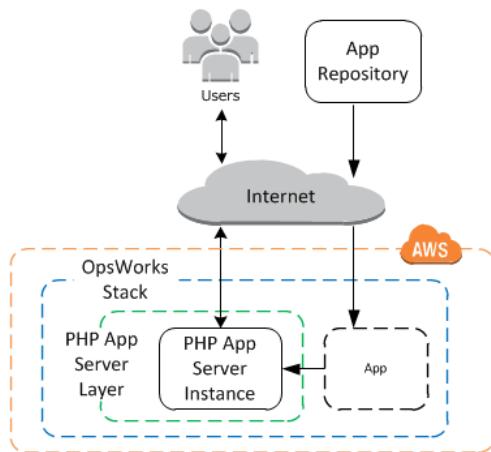
1. Go to <http://aws.amazon.com>, and then click **Sign Up**.
2. Follow the on-screen instructions.

Part of the sign-up procedure involves receiving a phone call and entering a PIN using the phone keypad.

After you have an AWS account, sign in to the [AWS OpsWorks](#) console. To have permissions to perform all of procedures in this walkthrough, you must sign in as an AWS OpsWorks *administrative user*—an IAM user with an attached policy that grants AWS OpsWorks Full Access permissions—or as the account owner. Alternatively, ask an administrative user to create a stack for you and assign your IAM user a Manage permissions level. For more information on how to grant AWS OpsWorks permissions, see [Managing User Permissions \(p. 277\)](#).

## Step 2: Create a Simple Application Server Stack

A basic application server stack consists of a single application server instance with a public IP address to receive user requests. Application code and any related files are stored in a separate repository and deployed from there to the server. The following diagram illustrates such a stack.



The stack has the following components:

- A *layer*, which represents a group of instances and specifies how they are to be configured.  
The layer in this example represents a group of PHP App Server instances.
- An *instance*, which represents an Amazon EC2 instance.  
In this case, the instance is configured to run a PHP app server. Layers can have any number of instances. AWS OpsWorks also supports several other app servers. For more information, see [Application Server Layers \(p. 81\)](#).
- An *app*, which contains the information required to install an application on the application server.  
The code is stored in a remote repository, such as Git repository or an Amazon S3 bucket.

The following sections describe how to use the AWS OpsWorks console to create the stack and deploy the application. You can also use an AWS CloudFormation template to provision a stack. For an example template that provisions the stack described in this topic, see [AWS OpsWorks Snippets](#).

### Topics

- [Step 1: Create a Stack \(p. 11\)](#)
- [Step 2: Add a PHP App Server Layer \(p. 13\)](#)
- [Step 3: Add an Instance to the PHP App Server Layer \(p. 14\)](#)
- [Step 4: Create and Deploy an App \(p. 16\)](#)

## Step 1: Create a Stack

You start an AWS OpsWorks project by creating a stack, which acts as a container for your instances and other resources. The stack configuration specifies some basic settings, such as the AWS region and the default operating system, that are shared by all the stack's instances.

### To create a new stack

1. **Add a Stack**

Sign into the [AWS OpsWorks console](#). If the account has no existing stacks, you will see the **Welcome to AWS OpsWorks** page; click **Add your first stack**. Otherwise, you will see the AWS OpsWorks dashboard, which lists your account's stacks; click **Add Stack**.



## 2. Configure the Stack

On the **Add Stack** page, specify the following settings:

### Name

Enter a name for your stack, which can contain alphanumeric characters (a–z, A–Z, and 0–9), and hyphens (-). The example stack for this walkthrough is named **MyStack**.

### IAM role

Specify the stack's IAM role. AWS OpsWorks needs to access other AWS services to perform tasks such as creating and managing Amazon EC2 instances. **IAM role** specifies an IAM (AWS Identity and Access Management) role that AWS OpsWorks assumes to work with other AWS services on your behalf. For more information, see [Allowing AWS OpsWorks to Act on Your Behalf \(p. 293\)](#).

- If your account has an existing AWS OpsWorks IAM role, you can select it from the list.

If the role was created by AWS OpsWorks, it will be named `aws-opsworks-service-role`.

- Otherwise, select **New IAM Role** to direct AWS OpsWorks to create a new role for you with the correct permissions.

**Note:** If you have AWS OpsWorks Full Access permissions, creating a new role requires several additional IAM permissions. For more information, see [Example Policies \(p. 286\)](#).

## Add Stack

Name	<input type="text" value="MyStack"/>
Region	US East (N. Virginia) <input type="button" value="▼"/>
VPC	No VPC <input type="button" value="▼"/>
Default Availability Zone	us-east-1a <input type="button" value="▼"/>
Default operating system <small>NEW</small>	Amazon Linux <input type="button" value="▼"/> Need a different OS? <a href="#">Let us know.</a>
Default root device type	<input checked="" type="radio"/> Instance store <input type="radio"/> EBS backed
IAM role	<input type="button" value="New IAM role"/>
Default SSH key	<input type="button" value="Do not use a default SSH key"/>
Default IAM instance profile	<input type="button" value="aws-opsworks-ec2-role"/>
Hostname theme	<input type="button" value="Layer Dependent"/>
Stack color	<div style="background-color: #333; width: 15px; height: 15px;"></div> <div style="background-color: #0070C0; width: 15px; height: 15px;"></div> <div style="background-color: #00AEEF; width: 15px; height: 15px;"></div> <div style="background-color: #2ECC71; width: 15px; height: 15px;"></div> <div style="background-color: #E67E22; width: 15px; height: 15px;"></div> <div style="background-color: #D35400; width: 15px; height: 15px;"></div> <div style="background-color: #9B59B6; width: 15px; height: 15px;"></div>

[NEW DEFAULT](#) [Advanced »](#)

[Cancel](#) [Add Stack](#)

Accept the default values for the other settings and click **Add Stack**. For more information on the various stack settings, see [Create a New Stack \(p. 41\)](#).

## Step 2: Add a PHP App Server Layer

Although a stack is basically a container for instances, you don't add instances directly to a stack. You add a layer, which represents a group of related instances, and then add instances to the layer.

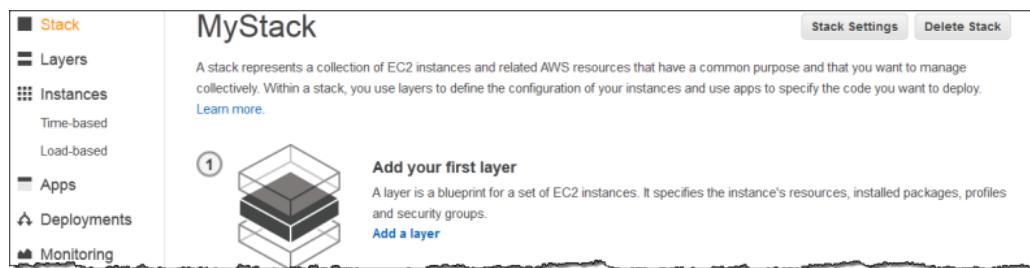
A layer is basically a blueprint that AWS OpsWorks uses to create set of Amazon EC2 instances with the same configuration. You add one layer to the stack for each group of related instances. AWS OpsWorks includes a set of built-in layers to represent groups of instances running standard software packages such as a MySQL database server or a PHP application server. In addition, you can create partially or fully customized layers to suit your specific requirements. For more information, see [Customizing AWS OpsWorks \(p. 230\)](#).

MyStack has one layer, the built-in PHP App Server layer, which represents a group of instances that function as PHP application servers. For more information, including descriptions of the built-in layers, see [Layers \(p. 57\)](#).

### To add a PHP App Server layer to MyStack

#### 1. Open the Add Layer Page

After you finish creating the stack, AWS OpsWorks displays the **Stack** page. Click **Add a layer** to add your first layer.



#### 2. Specify a Layer Type and Configure the Layer

In the **Layer type** box, select **PHP App Server**, accept the default **Elastic Load Balancer** setting and click **Add Layer**. After you create the layer, you can specify other attributes such as the EBS volume configuration by [editing the layer \(p. 59\)](#).

## Step 3: Add an Instance to the PHP App Server Layer

An AWS OpsWorks instance represents a particular Amazon EC2 instance:

- The instance's configuration specifies some basics like the Amazon EC2 operating system and size; it runs but doesn't do very much.
- The instance's layer adds functionality to the instance by determining which packages are to be installed, whether the instance has an Elastic IP address, and so on.

AWS OpsWorks installs an agent on each instance that interacts with the service. To add a layer's functionality to an instance, AWS OpsWorks directs the agent to run small applications called [Chef recipes](#), which can install applications and packages, create configuration files, and so on. AWS OpsWorks runs recipes at key points in the instance's [lifecycle \(p. 176\)](#). For example, OpsWorks runs Setup recipes after the instance has finished booting to handle tasks such as installing software, and runs Deploy recipes when you deploy an app to install the code and related files.

**Tip**

If you are curious about how the recipes work, all of the AWS OpsWorks built-in recipes are in a public GitHub repository: [OpsWorks Cookbooks](#). You can also create your own custom recipes and have AWS OpsWorks run them, as described later.

To add a PHP application server to MyStack, add an instance to the PHP App Server layer that you created in the previous step.

### To add an instance to the PHP App Server layer

1. **Open Add an Instance**

After you finish adding the layer, AWS OpsWorks displays the **Layers** page. Click **Instances** in the navigation pane and under **PHP App Server**, click **Add an instance**.

2. **Configure the Instance**

Each instance has a default host name that is generated for you by AWS OpsWorks. In this example, AWS OpsWorks simply adds a number to the layer's short name. You can configure each instance separately, including overriding some of the default settings that you specified when creating the stack, such as the Availability Zone or operating system. For this walkthrough, just accept the default settings and click **Add Instance** to add the instance to the layer. For more information, see [Instances \(p. 101\)](#).

## Instances

An instance represents an EC2 instance. Each instance belongs to a layer that defines the instance's settings, resources, installed packages, profiles and security groups. When you start the instance, OpsWorks uses the associated layer's blueprint to create and configure a corresponding EC2 instance. [Learn more](#).

### PHP App Server

No instances. [Add an instance](#).

[New](#) [Existing](#)

Hostname:

Size:

Availability Zone:

[Advanced »](#)

[Cancel](#) [Add Instance](#)

### 3. Start the Instance

So far, you have just specified the instance's configuration. You have to start an instance to create a running Amazon EC2 instance. AWS OpsWorks then uses the configuration settings to launch an Amazon EC2 instance in the specified Availability Zone. The details of how you start an instance depend on the instance's *scaling type*. In the previous step, you created an instance with the default scaling type, 24/7, which must be manually started and then runs until it is manually stopped. You can also create time-based and load-based scaling types, which AWS OpsWorks automatically starts and stops based on a schedule or the current load. For more information, see [Managing Load with Time-based and Load-based Instances \(p. 113\)](#).

Go to **php-app1** under **PHP App Server** and click **start** in the row's **Actions** column to start the instance.

## Instances

[Start All Instances](#)

An instance represents an EC2 instance. Each instance belongs to a layer that defines the instance's settings, resources, installed packages, profiles and security groups. When you start the instance, OpsWorks uses the associated layer's blueprint to create and configure a corresponding EC2 instance. [Learn more](#).



### PHP App Server

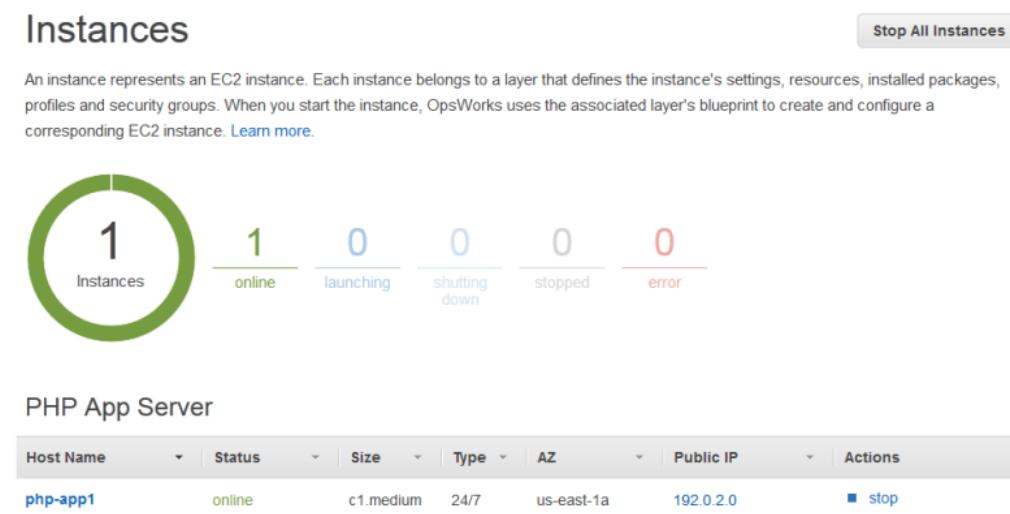
Host Name	Status	Size	Type	AZ	Public IP	Actions
php-app1	stopped	c1.medium	24/7	us-east-1a		<a href="#">start</a> <a href="#">delete</a>
<a href="#">+ Instance</a>						

#### 4. Monitor the Instance's Status during Startup

It typically takes a few minutes to boot the Amazon EC2 instance and install the packages. As startup progresses, the instance's **Status** field displays the following series of values:

1. **requested** - AWS OpsWorks has called the Amazon EC2 service to create the Amazon EC2 instance.
2. **pending** - AWS OpsWorks is waiting for the Amazon EC2 instance to start.
3. **booting** - The Amazon EC2 instance is booting.
4. **running\_setup** - The AWS OpsWorks agent is running the layer's Setup recipes, which handle tasks such as configuring and installing packages, and the Deploy recipes, which deploy any apps to the instance.
5. **online** - The instance is ready for use.

After php-app1 comes online, the **Instances** page should look like the following:



The page begins with a quick summary of all your stack's instances. Right now, it shows one online instance. In the php-app1 **Actions** column, notice that **stop**, which stops the instance, has replaced **start** and **delete**.

## Step 4: Create and Deploy an App

To make MyStack more useful, you need to deploy an app to the PHP App Server instance. You store an app's code and any related files in a repository, such as Git. You need to take a couple of steps to get those files to your application servers:

1. Create an app.

An app contains the information that AWS OpsWorks needs in order to download the code and related files from the repository. You can also specify additional information such as the app's domain.

2. Deploy the app to your application servers.

When you deploy an app, AWS OpsWorks triggers a Deploy lifecycle event. The agent then runs the instance's Deploy recipes, which download the files to the appropriate directory along with related tasks such as configuring the server, restarting the service, and so on.

### Note

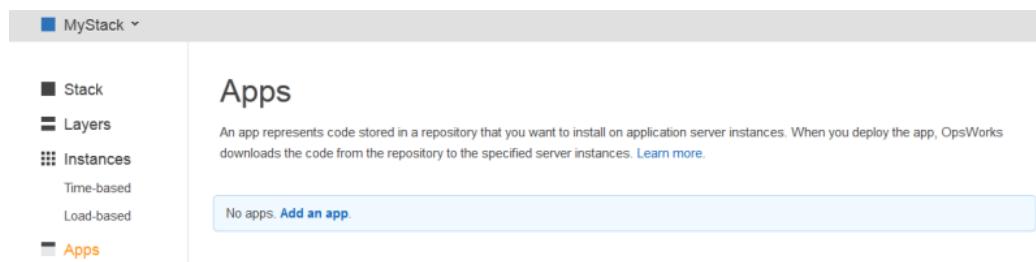
When you create a new instance, AWS OpsWorks automatically deploys any existing apps to the instance. However, when you create a new app or update an existing one, you must manually deploy the app or update to all existing instances.

This step shows how to manually deploy an example app from a public Git repository to an application server. If you would like to examine the application, go to <https://github.com/amazonwebservices/opsworks-demo-php-simple-app>. The application used in this example is in the version1 branch. AWS OpsWorks also supports several other repository types. For more information, see [Application Source \(p. 129\)](#).

### To create and deploy an app

#### 1. Open the Apps Page

In the navigation pane, click **Apps** and on the **Apps** page, click **Add an app**.



#### 2. Configure the App

On the **App** page, specify the following values:

##### Name

The app's name, which AWS OpsWorks uses for display purposes. The example app is named `SimplePHPApp`. AWS OpsWorks also generates a short name—`simplephapp` for this example—that is used internally and by the Deploy recipes, as described later.

##### App type

The app's type, which determines where to deploy the app. The example uses **PHP**, which deploys the app to PHP App Server instances.

##### Data source type

An associated database server. For now, select **None**; we'll introduce database servers in [Step 3: Add a Back-end Data Store \(p. 20\)](#).

##### Repository type

The app's repository type. The example app is stored in a **Git** repository.

##### Repository URL

The app's repository URL. The example URL is: `git://github.com/amazonwebservices/opsworks-demo-php-simple-app.git`

##### Branch/Revision

The app's branch or version. This part of the walkthrough uses the `version1` branch.

Keep the default values for the remaining settings and click **Add App**. For more information, see [Adding Apps \(p. 125\)](#).

The screenshot shows the 'Add App' configuration dialog. It has three main sections: 'Settings', 'Data Sources', and 'Application Source'. In the 'Settings' section, 'Name' is set to 'SimplePHPApp' and 'Type' is set to 'PHP'. Under 'Data Sources', 'Data source type' is set to 'None'. In the 'Application Source' section, 'Repository type' is set to 'Git', 'Repository URL' is 'git://github.com/amazonwebservices/o...', and 'Branch/Revision' is 'version1'.

### 3. Open the Deployment Page

To install the code on the server, you must *deploy* the app. To do so, click **deploy** in the SimplePH-PApp **Actions** column.

The screenshot shows the 'Apps' list page. It displays a single app entry: 'SimplePHPApp' of type 'php'. Below the table is a '+ App' button.

Name	Type	Last deployment	Actions
SimplePHPApp	php		<a href="#">deploy</a> <a href="#">edit</a> <a href="#">delete</a>

### 4. Deploy the App

When you deploy an app, the agent runs the Deploy recipes on the PHP App Server instance, which download and configure the application.

**Command** should already be set to **deploy**. Keep the defaults for the other settings and click **Deploy** to deploy the app.

## Deploy app

### Settings

App	SimplePHPApp
Command	deploy
Comment	Optional

[Advanced »](#)

### Instances (1)

OpsWorks will run this command on **1 of 1** instances. The assigned recipes are run on all selected instances.

- PHP App Server  php-app1 (online)  
Click to select all instances in this layer

[Cancel](#) [Deploy](#)

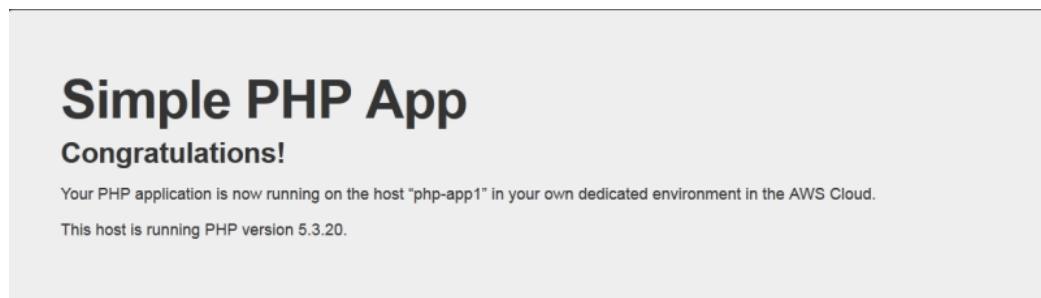
When deployment is complete, the **Deployment** page displays a **Status of Successful**, and **php-app1** will have a green check mark next to it.

### 5. Run SimplePHPApp

SimplePHPApp is now installed and ready to go. To run it, click **Instances** in the navigation pane to go to the **Instances** page. Then click the php-app1 instance's public IP address.

PHP App Server						
Host Name	Status	Size	Type	AZ	Public IP	Actions
php-app1	online	c1.medium	24/7	us-east-1a	198.51.100.0	<a href="#">stop</a>
<a href="#">+ Instance</a>						

You should see the following in your browser.

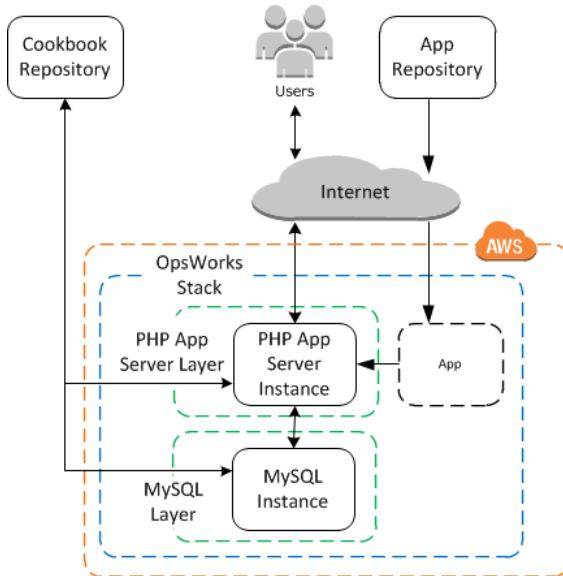


### Tip

This walkthrough assumes that you will go on to the next section and ultimately complete the entire walkthrough in one session. If you prefer, you can stop at any point and continue later by signing in to AWS OpsWorks and opening the stack. However, you are charged for any AWS resources that you use, such as online instances. To avoid unnecessary charges, you can stop your instance, which terminates the corresponding EC2 instance. You can start the instances again when you are ready to continue.

## Step 3: Add a Back-end Data Store

Step 1: Sign in to the AWS OpsWorks Console (p. 10) showed you how to create a stack that served a PHP application. However, that was a very simple application that did little more than display some static text. Production applications commonly use a back-end data store, yielding a stack configuration something like the illustration that follows.



This section shows how to extend MyStack to include a back-end MySQL database server. You need to do more than just add a MySQL server to the stack, though. You also have to configure the app to communicate properly with the database server. AWS OpsWorks doesn't do this for you; you will need to implement some custom recipes to handle that task.

### Topics

- [Step 1: Add a Back-end Database \(p. 20\)](#)
- [Step 2: Update SimplePHPApp \(p. 21\)](#)
- [A Short Digression: Cookbooks, Recipes, and AWS OpsWorks Attributes \(p. 23\)](#)
- [Step 3: Add the Custom Cookbooks to MyStack \(p. 28\)](#)
- [Step 4: Run the Recipes \(p. 28\)](#)
- [Step 5: Deploy SimplePHPApp, Version 2 \(p. 29\)](#)
- [Step 6: Run SimplePHPApp \(p. 30\)](#)

## Step 1: Add a Back-end Database

The new version of SimplePHPApp stores its data in a back-end database. AWS OpsWorks supports two types of database servers:

- The [MySQL AWS OpsWorks layer \(p. 80\)](#) is a blueprint for creating Amazon EC2 instances that host a MySQL database master.
- The Amazon RDS service layer provides a way to incorporate an [Amazon RDS instance](#) into a stack.

You can also use other databases, such as Amazon DynamoDB, or create a custom layer to support databases such as [MongoDB](#). For more information, see the section called “[Using Other Back-end Data Stores](#)” (p. 300).

This example uses a MySQL layer.

### To add a MySQL layer to MyStack

1. On the **Layers** page, click **+ Layer**.
2. On the **Add Layer** page, for **Layer type**, select **MySQL**, accept the default settings, and click **Add Layer**.

**Add Layer**

OpsWorks    RDS

Layer type: MySQL Looking for a different Layer type? [Let us know](#).  
A MySQL Master layer is a blueprint for instances that function as MySQL relational database servers. [Learn more](#).

MySQL root user password: d8uvfija3q

Set root user password on every instance: Yes

Cancel    Add Layer

### To add an instance to the MySQL layer

1. On the **Layers** page's **MySQL** row, click **Add an instance**.
2. On the **Instances** page, under **MySQL**, click **Add an instance**.
3. Accept the defaults and click **Add instance**, but don't start it yet.

#### Note

AWS OpsWorks automatically creates a database named using the app's short name, simplephpapp for this example. You'll need this name if you want to use [Chef recipes](#) to interact with the database.

## Step 2: Update SimplePHPApp

To start, you need a new version of SimplePHPApp that uses a back-end data store. With AWS OpsWorks, it's easy to update an application. If you use a Git or Subversion repository, you can have a separate repository branch for each app version. The example app stores a version of the app that uses a back-end database in the Git repository's `version2` branch. You just need to update the app's configuration to specify the new branch and redeploy the app.

### To update SimplePHPApp

1. **Open the App's Edit Page**

In the navigation pane, click **Apps** and then click **edit** in the **SimplePHPApp** row's **Actions** column.

2. **Update the App's Configuration**

Change the following settings.

#### Branch/Revision

This setting indicates the app's repository branch. The first version of SimplePHPApp didn't connect to a database. To use a the database-enabled version of the app, set this value to **version2**.

#### Document root

This setting specifies your app's root folder. The first version of SimplePHPApp used the default setting, which installs `index.php` in the server's standard root folder (`/srv/www` for PHP apps). If you specify a subfolder here—just the name, no leading '`/`'—AWS OpsWorks appends it to the standard folder path. Version 2 of SimplePHPApp should go in `/srv/www/web`, so set **Document root** to `web`.

#### Data source type

This setting associates a database server with the app. The example uses the MySQL instance that you created in the previous step, so set **Data source type** to OpsWorks and **Database instance** to the instance you created in the previous step, **db-master1 (mysql)**. Leave **Database name** empty; AWS OpsWorks will create a database on the server named with the app's short name, `simplephpapp`.

Then click **Save** to save the new configuration.

The screenshot shows the 'Add App' configuration dialog. The 'Settings' section includes fields for Name (SimplePHP), Type (PHP), and Document root (web). The 'Data Sources' section shows Data source type (OpsWorks, selected), Database instance (db-master1 (mysql)), and Database name (empty). The 'Application Source' section includes Repository type (Git), Repository URL (git://github.com/amazonwebservices/oj), Repository SSH key (Optional), and Branch/Revision (version2). The 'version2' field is circled in red. At the bottom left is a link to 'Add Domains'.

3. Start the MySQL instance.

After you update an app, AWS OpsWorks automatically deploys the new app version to any new app server instances when you start them. However, AWS OpsWorks does not automatically deploy the new app version to existing server instances; you must do that manually, as described in [Step 4: Create and Deploy an App \(p. 16\)](#). You could deploy the updated SimplePHPApp now, but for this example, it's better to wait a bit.

## A Short Digression: Cookbooks, Recipes, and AWS OpsWorks Attributes

You now have app and database servers, but they aren't quite ready to use. You still need to set up the database and configure the app's connection settings. AWS OpsWorks doesn't handle these tasks automatically, but it does support Chef cookbooks, recipes, and dynamic attributes. You can implement a pair of recipes, one to set up the database and one to configure the app's connection settings, and have AWS OpsWorks run them for you.

The `phpapp` cookbook, which contains the required recipes, is already implemented and ready for use; you can just skip to [Step 3: Add the Custom Cookbooks to MyStack \(p. 28\)](#) if you prefer. If you'd like to know more, this section provides some background on cookbooks and recipes and describes how the recipes work. To see the cookbook itself, go to the [phpapp cookbook](#).

### Topics

- [Recipes and Attributes \(p. 23\)](#)
- [Set Up the Database \(p. 25\)](#)
- [Connect the Application to the Database \(p. 26\)](#)

## Recipes and Attributes

A Chef recipe is basically a specialized Ruby application that performs tasks on an instance such as installing packages, creating configuration files, executing shell commands, and so on. Groups of related recipes are organized into *cookbooks*, which also contain supporting files such as templates for creating configuration files.

AWS OpsWorks has a set of cookbooks that support the built-in layers. You can also create custom cookbooks with your own recipes to perform custom tasks on your instances. This topic provides a brief introduction to recipes and shows how to use them to set up the database and configure the app's connection settings. For more information on cookbooks and recipes, see [Cookbooks and Recipes \(p. 153\)](#) or [Customizing AWS OpsWorks \(p. 230\)](#).

Recipes usually depend on Chef *attributes* for input data:

- Some of these attributes are defined by Chef and provide basic information about the instance such as the operating system.
- AWS OpsWorks defines a set of attributes that contain information about the stack—such as the layer configurations—and about deployed apps—such as the app repository.

You can add custom attributes to this set by assigning [custom JSON \(p. 53\)](#) to the stack or deployment.

- Your cookbooks can also define attributes, which are specific to the cookbook.

The `phpapp` cookbook attributes are defined in `attributes/default.rb`.

For a complete list of AWS OpsWorks attributes, see [Appendix C: AWS OpsWorks Attribute Reference \(p. 378\)](#). For more information, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

Attributes are organized in a hierarchical structure, which can be represented as a JSON object. The following example shows a JSON representation of the deployment attributes for `SimplePHPApp`, which includes the values that you need for the task at hand.

```
"deploy": {
    "simplephpapp": {
        "sleep_before_restart": 0,
        "rails_env": null,
        "document_root": "web",
        "deploy_to": "/srv/www/simplephpapp",
        "ssl_certificate_key": null,
        "ssl_certificate": null,
        "deploying_user": null,
        "ssl_certificate_ca": null,
        "ssl_support": false,
        "migrate": false,
        "mounted_at": null,
        "application": "simplephpapp",
        "auto_bundle_on_deploy": true,
        "database": {
            "reconnect": true,
            "database": "simplephpapp",
            "host": null,
            "adapter": "mysql",
            "data_source_provider": "stack",
            "password": "dlzethv0sm",
            "port": 3306,
            "username": "root"
        },
        "scm": {
            "revision": "version2",
            "scm_type": "git",
            "user": null,
            "ssh_key": null,
            "password": null,
            "repository": "git://github.com/amazonwebservices/opsworks-demo-php-simple-app.git"
        },
        "symlink_before_migrate": {
            "config/opsworks.php": "opsworks.php"
        },
        "application_type": "php",
        "domains": [
            "simplephpapp"
        ],
        "memcached": {
            "port": 11211,
            "host": null
        },
        "symlinks": {
        },
        "restart_command": "echo 'restarting app'"
    }
}
```

You incorporate this data into your application by using Chef node syntax, like the following:

```
[:deploy][:simplephpapp][:database][:username]
```

The `deploy` node has a single app node, `simplephpapp`, that contains information about the app's database, Git repository, and so on. The example represents the value of the database user name, which resolves to `root`.

## Set Up the Database

The MySQL layer's built-in Setup recipes automatically create a database for the app named with the app's shortname, so for this example you already have a database named `simplephpapp`. However, you need to finish the setup by creating a table for the app to store its data. You could create the table manually, but a better approach is to implement a custom recipe to handle the task, and have AWS OpsWorks run it for you. This section describes how the recipe, `dbsetup.rb`, is implemented. The procedure for having AWS OpsWorks run the recipe is described later.

To see the recipe in the repository, go to [dbsetup.rb](#). The following example shows the `dbsetup.rb` code.

```
node[:deploy].each do |app_name, deploy|
  execute "mysql-create-table" do
    command "/usr/bin/mysql -u#{deploy[:database][:username]} -p#{deploy[:database][:password]} #{deploy[:database][:database]} -e'CREATE TABLE
#{node[:phpapp][:dbtable]}(
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  author VARCHAR(63) NOT NULL,
  message TEXT,
  PRIMARY KEY (id)
)'"
    not_if "/usr/bin/mysql -u#{deploy[:database][:username]} -p#{deploy[:database][:password]} #{deploy[:database][:database]} -e'SHOW TABLES' | grep
#{node[:phpapp][:dbtable]}"
    action :run
  end
end
```

This recipe iterates over the apps in the `deploy` node. This example has only one app, but it's possible for a `deploy` node to have multiple apps.

`execute` is a *Chef resource* that executes a specified command. In this case, it's a MySQL command that creates a table. The `not_if` directive ensures that the command does not run if the specified table already exists. For more information on Chef resources, see [Resources and Providers Reference](#).

The recipe inserts attribute values into the command string, using the node syntax discussed earlier. For example, the following inserts the database's user name.

```
#{deploy[:database][:username]}
```

Let's unpack this somewhat cryptic code:

- For each iteration, `deploy` is set to the current app node, so it resolves to `[:deploy][:app_name]`. For this example, it resolves to `[:deploy][:simplephpapp]`.
- Using the deployment attribute values shown earlier, the entire node resolves to `root`.
- You wrap the node in `#{}`  to insert it into a string.

Most of the other nodes resolve in a similar way. The exception is `# {node[:phpapp][:dbtable]}`, which is defined by the custom cookbook's attributes file and resolves to the table name, `urler`. The actual command that runs on the MySQL instance is therefore:

```
" /usr/bin/mysql
  -uroot
  -pvjud1hw5v8
  simplephpapp
  -e'CREATE TABLE urler(
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    author VARCHAR(63) NOT NULL,
    message TEXT,
    PRIMARY KEY (id))'
"
```

This command creates a table named `urler` with `id`, `author`, and `message` fields, using the credentials and database name from the deployment attributes.

## Connect the Application to the Database

The second piece of the puzzle is the application, which needs connection information such as the database password to access the table. SimplePHPApp effectively has only one working file, `app.php`; all `index.php` does is load `app.php`.

`app.php` includes `db-connect.php`, which handles the database connection, but that file is not in the repository. You can't create `db-connect.php` in advance because it defines the database based on the particular instance. Instead, the `appsetup.rb` recipe generates `db-connect.php` using connection data from the deployment attributes.

To see the recipe in the repository, go to [appsetup.rb](#). The following example shows the `appsetup.rb` code.

```
node[:deploy].each do |app_name, deploy|
  script "install_composer" do
    interpreter "bash"
    user "root"
    cwd "#{deploy[:deploy_to]}/current"
    code <<-EOH
      curl -s https://getcomposer.org/installer | php
      php composer.phar install
    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
  end

```

```
variables(
  :host =>      (deploy[:database][:host] rescue nil),
  :user =>       (deploy[:database][:username] rescue nil),
  :password =>  (deploy[:database][:password] rescue nil),
  :db =>         (deploy[:database][:database] rescue nil),
  :table =>     (node[:phpapp][:dbtable] rescue nil)
)

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

Like `dbsetup.rb`, `appsetup.rb` iterates over apps in the `deploy` node—just `simplephpapp` again. It runs a code block with a `script` resource and a `template` resource.

The `script` resource installs [Composer](#)—a dependency manager for PHP applications. It then runs Composer's `install` command to install the dependencies for the sample application to the app's root directory.

The `template` resource generates `db-connect.php` and puts it in `/srv/www/simplephpapp/current`. Note the following:

- The recipe uses a conditional statement to specify the file owner, which depends on the instance's operating system.
- The `only_if` directive tells Chef to generate the template only if the specified directory exists.

A `template` resource operates on a template that has essentially the same content and structure as the associated file but includes placeholders for various data values. The `source` parameter specifies the template, `db-connect.php.erb`, which is in the `phpapp` cookbook's `templates/default` directory, and contains the following:

```
<?php
define('DB_NAME', '<%= @db%>');
define('DB_USER', '<%= @user%>');
define('DB_PASSWORD', '<%= @password%>');
define('DB_HOST', '<%= @host%>');
define('DB_TABLE', '<%= @table%>');
?>
```

When Chef processes the template, it replaces the `<%= =>` placeholders with the value of the corresponding variables in the template resource, which are in turn drawn from the deployment attributes. The generated file is therefore:

```
<?php
define('DB_NAME', 'simplephpapp');
define('DB_USER', 'root');
define('DB_PASSWORD', 'ujn12pw');
define('DB_HOST', '10.214.175.110');
define('DB_TABLE', 'urler');
?>
```

## Step 3: Add the Custom Cookbooks to MyStack

You store custom cookbooks in a repository, much like apps. Each stack can have a repository that contains one set of custom cookbooks. You then direct AWS OpsWorks to install your custom cookbooks on the stack's instances.

1. Click **Stack** in the navigation pane to see the page for the current stack.
2. Click **Stack Settings**, and then click **Edit**.
3. Modify the stack configuration as follows:
  - **Use custom Chef Cookbooks** – Yes
  - **Repository type** – Git
  - **Repository URL** – `git://github.com/amazonwebservices/opsworks-example-cookbooks.git`
4. Click **Save** to update the stack configuration.



AWS OpsWorks then installs the contents of your cookbook repository on all of the stack's instances. If you create new instances, AWS OpsWorks automatically installs the cookbook repository.

### Note

If you need to update any of your cookbooks, or add new cookbooks to the repository, you can do so without touching the stack settings. AWS OpsWorks will automatically install the updated cookbooks on all new instances. However, AWS OpsWorks does not automatically install updated cookbooks on the stack's online instances. You must explicitly direct AWS OpsWorks to update the cookbooks by running the `Update Cookbooks` stack command. For more information, see [Run Stack Commands \(p. 52\)](#).

## Step 4: Run the Recipes

After you have your custom cookbook, you need to run the recipes on the appropriate instances. You could [run them manually \(p. 178\)](#). However, recipes typically need to be run at predictable points in an instance's lifecycle, such as after the instance boots or when you deploy an app. This section describes a much simpler approach: have AWS OpsWorks automatically run them for you at the appropriate time.

AWS OpsWorks supports a set of [lifecycle events \(p. 176\)](#) that simplify running recipes. For example, the Setup event occurs after an instance finishes booting and the Deploy event occurs when you deploy an app. Each layer has a set of built-in recipes associated with each lifecycle event. When a lifecycle event occurs on an instance, the agent runs the associated recipes for each of the instance's layers. To have AWS OpsWorks run a custom recipe automatically, add it to the appropriate lifecycle event on the appropriate layer and the agent will run the recipe after the built-in recipes are finished.

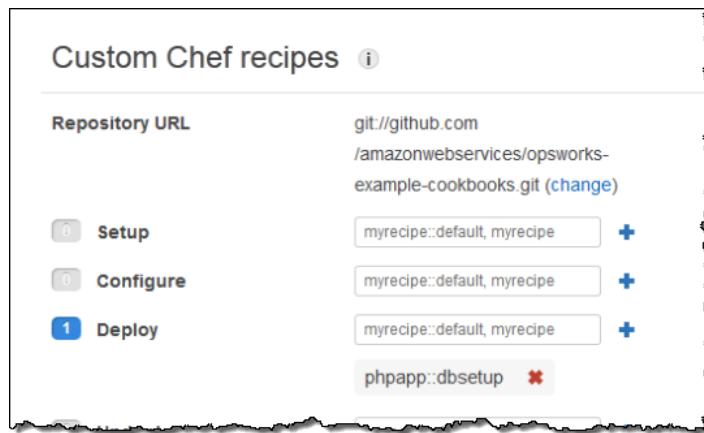
For this example, you need to run two recipes, `dbsetup.rb` on the MySQL instance and `appsetup.rb` on the PHP App Server instance.

**Note**

You specify recipes on the console by using the `cookbook_name::recipe_name` format, where `recipe_name` does not include the .rb extension. For example, you refer to `dbsetup.rb` as `phpapp::dbsetup`.

**To assign custom recipes to lifecycle events**

1. On the **Layers** page, for MySQL, click **Recipes** and then click **Edit**.
2. In the **Custom Chef recipes** section, enter [phpapp::dbsetup \(p. 25\)](#) for **Deploy**.



3. Click the + icon to assign the recipe to the event and click **Save** to save the new layer configuration.
4. Return to the **Layers** page and repeat the procedure to assign `phpapp::appsetup` to the **PHP App Server** layer's **Deploy** event.

## Step 5: Deploy SimplePHPApp, Version 2

The final step is to deploy the new version of SimplePHPApp.

**To deploy SimplePHPApp**

1. On the **Apps** page, click **deploy** in the **SimplePHPApp** app's **Actions**.

### Apps

An app represents code stored in a repository that you want to install on application server instances. When you deploy the app, OpsWorks downloads the code from the repository to the specified server instances. [Learn more](#).

Name	Type	Last deployment	Actions
SimplePHPApp	php	2013-02-19 21:34:43 UTC	
<a href="#">+ App</a>			

2. Accept the defaults and click **Deploy**.

## Deploy App

Settings

App	SimplePHPApp
Command	Deploy
Comment	Optional

[Advanced »](#)

Instances ⓘ

OpsWorks will run this command on **2 of 2** instances. The assigned recipes are run on all selected instances.

<input checked="" type="checkbox"/> PHP App Server	<input checked="" type="checkbox"/> php-app1 ● Click to select instances in this layer
<input checked="" type="checkbox"/> MySQL	<input checked="" type="checkbox"/> db-master1 ● Click to select instances in this layer

[Cancel](#) [Deploy](#)

When you click **Deploy** on the **Deploy App** page, you trigger a Deploy lifecycle event, which notifies the agents to run their Deploy recipes. By default, you trigger the event on all of the stack's instances. The built-in Deploy recipes deploy the app only to the appropriate instances for the app type, PHP App Server instances in this case. However, it is often useful to trigger the Deploy event on other instances, to allow them to respond to the app deployment. In this case, you also want to trigger Deploy on the MySQL instance to set up the database.

Note the following:

- The agent on the PHP App Server instance runs the layer's built-in recipe, followed by `appsetup.rb`, which configures the app's database connection.
- The agent on the MySQL instance doesn't install anything, but it runs `dbsetup.rb` to create the `urler` table.

When the deployment is complete, the **Status** will change to **successful** on the **Deployment** page.

## Step 6: Run SimplePHPApp

After the deployment status changes to **successful**, you can run the new SimplePHPApp version, as follows.

### To run SimplePHPApp

- On the **Instances** page, click the public IP address in the **php-app1** row.

You should see the following page in your browser.



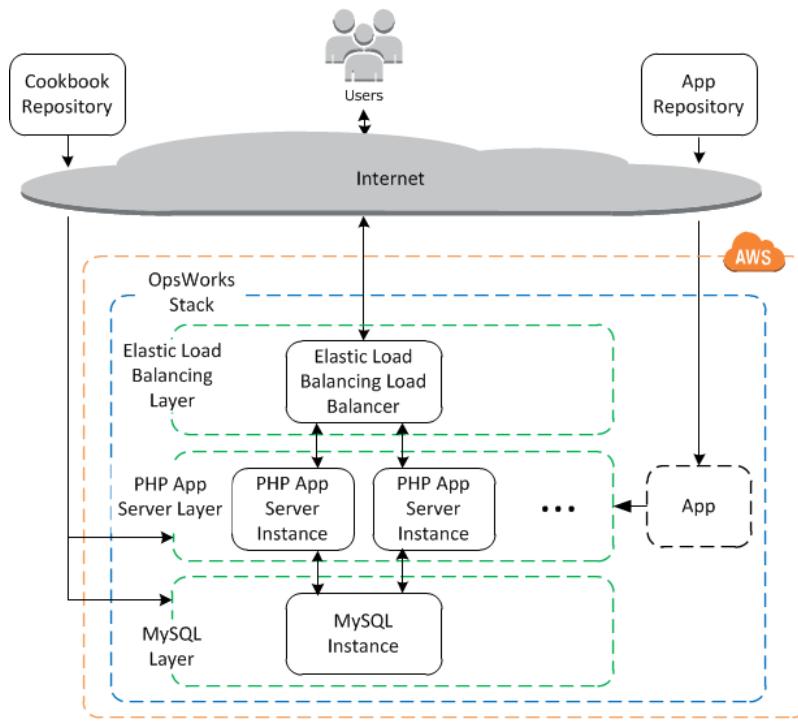
2. Click **Share Your Thought** and type something like `Hello world!` for **Your Thought** and your name for **Your Name**. Then click the **Submit Your Thought** to add the message to the database.

The image shows a screenshot of a web form titled "Share Your Thought!". At the top, there is a green success message: "Success! Thank you for sharing your thought.". Below it are two input fields: "Your Thought" and "Your Name". At the bottom are two buttons: "✓ Submit Your Thought" and "Go Back".

3. Click **Go Back** to view all the messages in the database.

## Step 4: Scale Out MyStack

MyStack currently has only one application server. A production stack will probably need multiple application servers to handle the incoming traffic and a load balancer to distribute the incoming traffic evenly across the application servers. The architecture will look something like the following:



AWS OpsWorks makes it easy to scale out stacks. This section describes the basics of how to scale out a stack by adding a second 24/7 PHP App Server instance to MyStack and putting both instances behind an Elastic Load Balancing load balancer. You can easily extend the procedure to add an arbitrary number of 24/7 instances, or you can use time-based or load-based instances to have AWS OpsWorks scale your stack automatically. For more information, see [Managing Load with Time-based and Load-based Instances \(p. 113\)](#).

## Step 1: Add a Load Balancer

Elastic Load Balancing is an AWS service that automatically distributes incoming application traffic across multiple Amazon EC2 instances. In addition to distributing traffic, Elastic Load Balancing does the following:

- Detects unhealthy Amazon EC2 instances.  
It reroutes traffic to the remaining healthy instances until the unhealthy instances have been restored.
- Automatically scales request handling capacity in response to incoming traffic

### Tip

A load balancer can serve two purposes. The obvious one is to equalize the load on your application servers. In addition, many sites prefer to isolate their application servers and databases from direct user access. With AWS OpsWorks, you can do this by running your stack in a virtual private cloud (VPC) with a public and private subnet, as follows.

- Put the application servers and database in the private subnet, where they can be accessed by other instances in the VPC but not by users.
- Direct user traffic to a load balancer in the public subnet, which then forwards the traffic to the application servers in the private subnet and returns responses to users.

For more information, see [Running a Stack in a VPC \(p. 44\)](#). For an AWS CloudFormation template that extends the example in this walkthrough to run in a VPC, see [OpsWorksVP-CELB.template](#).

Although Elastic Load Balancing is often referred to as a layer, it works a bit differently than the other built-in layers. Instead of creating a layer and adding instances to it, you create an Elastic Load Balancing load balancer by using the Amazon EC2 console and then attach it to one of your existing layers, usually an application server layer. AWS OpsWorks then registers the layer's existing instances with the service and automatically adds any new instances. The following procedure describes how to add a load balancer to MyStack's PHP App Server layer.

### To attach a load balancer to the PHP App Server layer

1. Use the Amazon EC2 console to create a new load balancer for MyStack. The details depend on whether your account supports EC2 Classic. For more information, see [Get Started with Elastic Load Balancing](#). When you run the **Create Load Balancer** wizard, configure the load balancer as follows:

#### Define Load Balancer

Assign the load balancer an easily recognizable name, like PHP-LB, to make it easier to locate in the AWS OpsWorks console. Then click **Continue** to accept the defaults for the remaining settings.

#### Configure Health Check

Set the ping path to / and accept the defaults for the remaining settings.

#### Assign Security Groups

If your account supports default VPC, the wizard displays this page to determine the load balancer's security group. It does not display this page for EC2 Classic.

For this walkthrough, specify **default VPC security group**.

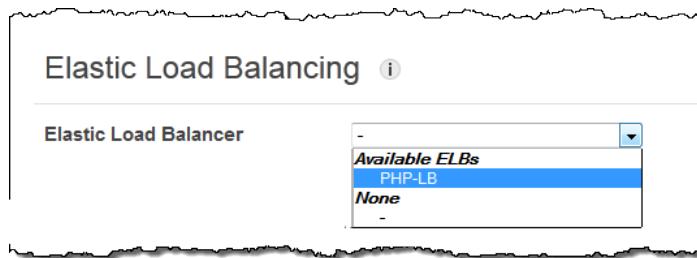
#### Add EC2 Instances

Click **Continue**; AWS OpsWorks automatically takes care of registering instances with the load balancer.

#### Review

Review your choices and click **Create** and then **Close**, which launches the load balancer.

2. If your account supports default VPC, after you launch the load balancer you must ensure that its security group has appropriate ingress rules. The default rule does not accept any inbound traffic.
  1. Click **Security Groups** in the Amazon EC2 navigation pane.
  2. Select **default VPC security group**
  3. Click **Edit** on the **Inbound** tab.
  4. For this walkthrough, set **Source** to **Anywhere**, which directs the load balancer to accept incoming traffic from any IP address.
3. Return to the AWS OpsWorks console. On the **Layers** page, click the layer's **Network** link and then click **Edit**.
4. Under **Elastic Load Balancing**, select the load balancer that you created in Step 1 and click **Save**.



After you have attached the load balancer to the layer, AWS OpsWorks automatically registers the layer's current instance's and adds new instances as they come online.

5. On the **Layers** page, click the load balancer's name to open its details page. When registration is complete and the instance passes a health check, AWS OpsWorks puts a green check next to the instance on the load balancer page.

## ELB PHP-LB

Elastic Load Balancing associates your load balancer with your EC2 instances using IP addresses. [Learn more.](#)

us-east-1c

php-app1

You can now run SimplePHPApp by sending a request to the load balancer.

### To run SimplePHPApp through the load balancer

1. Open load balancer's details page again, if it is not already open.
2. On the properties page, verify the instance's health-check status and click the load balancer's DNS name to run SimplePHPApp. The load balancer forwards the request to the PHP App Server instance and returns the response, which should look exactly the same as the response you get when you click the PHP App Server instance's public IP address.

## ELB PHP-LB

Elastic Load Balancing associates your load balancer with your EC2 instances using IP addresses. [Learn more.](#)

### Settings

Layer	PHP App Server
DNS Name	<a href="#">PHP-LB-541567884.us-east-1.elb.amazonaws.com</a>
Region	US East (N. Virginia)
Attached availability zones	us-east-1c

### Tip

AWS OpsWorks also supports the HAProxy load balancer, which might have advantages for some applications. For more information, see [HAProxy AWS OpsWorks Layer \(p. 72\)](#).

## Step 2: Add PHP App Server Instances

Now the load balancer is in place, you can scale out the stack by adding more instances to the PHP App Server layer. From your perspective, the operation is seamless. Each time a new PHP App Server instance comes online, AWS OpsWorks automatically registers it with the load balancer and deploys SimplePHPApp, so the server can immediately start handling incoming traffic. For brevity, this topic shows how to add one additional PHP App Server instance, but you can use the same approach to add as many as you need.

### To add another instance to the PHP App Server layer

1. On the Instances page, click **+ Instance** under **PHP App Server**.
2. Accept the default settings and click **Add Instance**.
3. Click **start** to start the instance.

**Instances** [Start All Instances](#)

An instance represents an EC2 instance. Each instance belongs to a layer that defines the instance's settings, resources, installed packages, profiles and security groups. When you start the instance, OpsWorks uses the associated layer's blueprint to create and configure a corresponding EC2 instance. [Learn more](#).

Instances	online	launching	shutting down	stopped	error
1	0	0	0	1	0

**PHP App Server**

Host Name	Status	Size	Type	AZ	Public IP	Actions
php-app1	stopped	c1.medium	24/7	us-east-1a		<a href="#">start</a> <a href="#">edit</a> <a href="#">delete</a>

[+ Instance](#)

## Step 3: Monitor MyStack

AWS OpsWorks uses Amazon CloudWatch to provide metrics for a stack and summarizes them for your convenience on the **Monitoring** page. You can view metrics for the entire stack, a specified layer, or a specified instance.

### To monitor MyStack

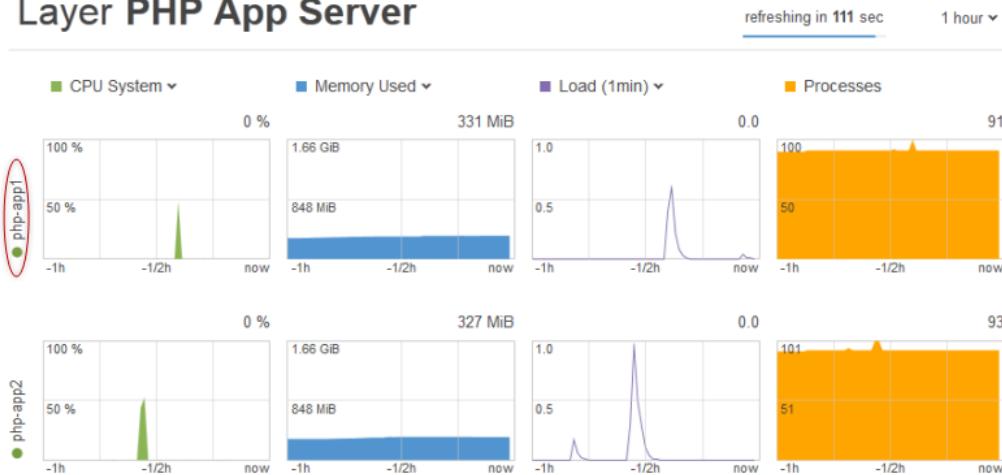
1. In the navigation pane, click **Monitoring**, which displays a set of graphs with average metrics for each layer. You can use the menus for **CPU System**, **Memory Used**, and **Load** to display different related metrics.

## Monitoring Layers

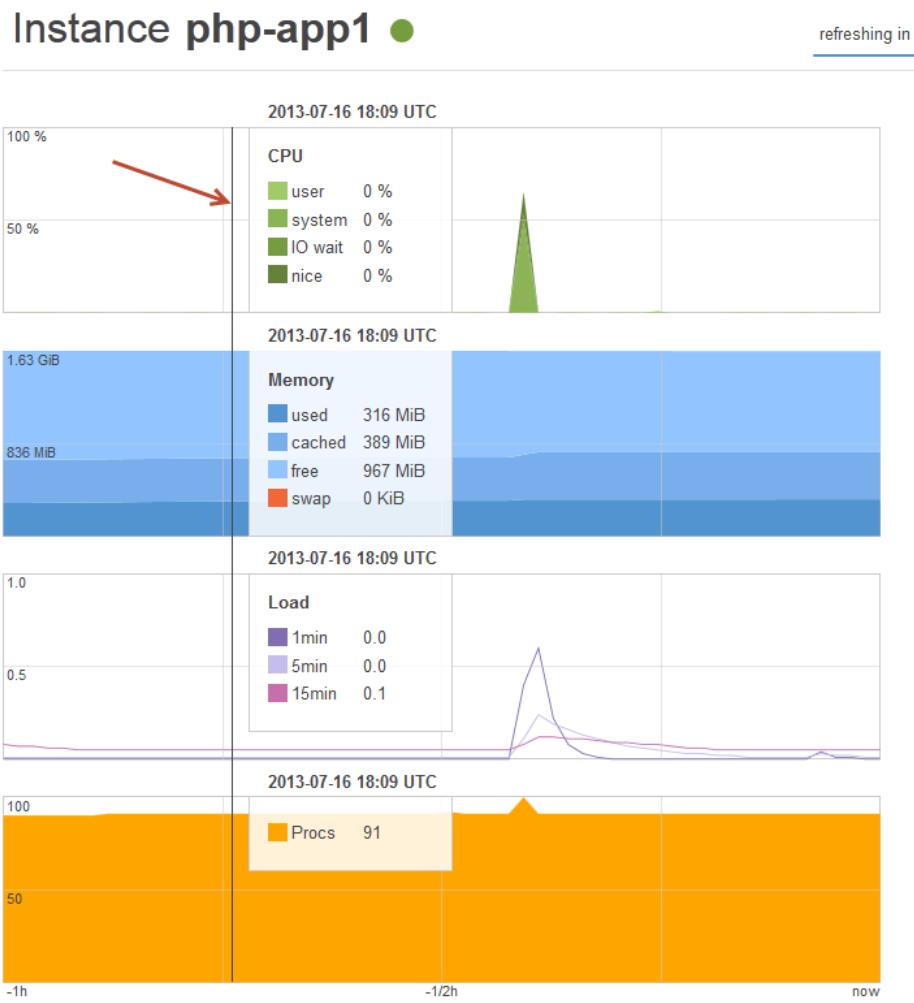


- Click **PHP App Server** to see metrics for each of the layer's instances.

## Layer PHP App Server



- Click **php-app1** to see metrics for that instance. You can see metrics for any particular point in time by moving the slider.



**Tip**

AWS OpsWorks also supports the Ganglia monitoring server, which might have advantages for some applications. For more information, see [Ganglia Layer \(p. 98\)](#).

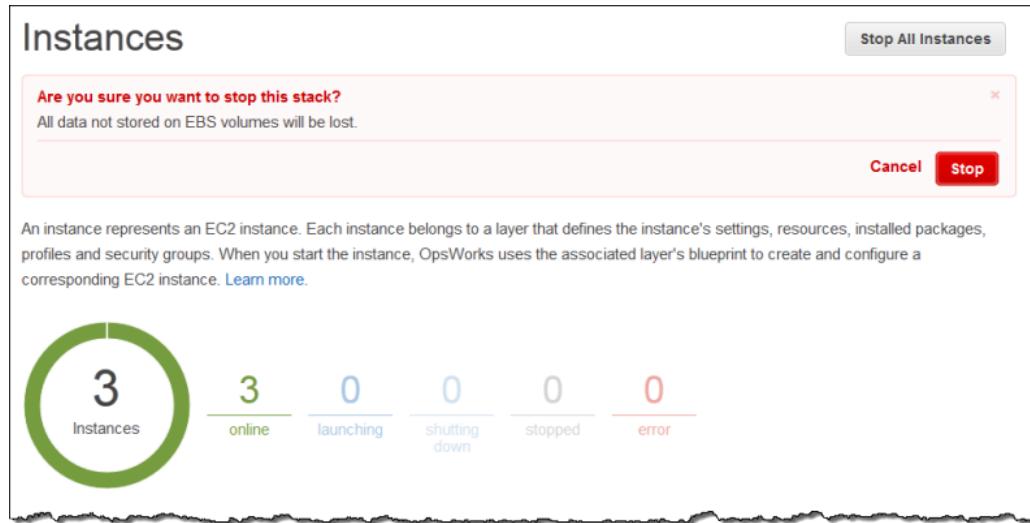
## Delete MyStack

As soon as you begin using AWS resources like Amazon EC2 instances you are charged based on your usage. If you are finished for now, you should stop the instances so that you do not incur any unwanted charges. If you don't need the stack anymore, you can delete it.

### To delete MyStack

#### 1. Stop all Instances

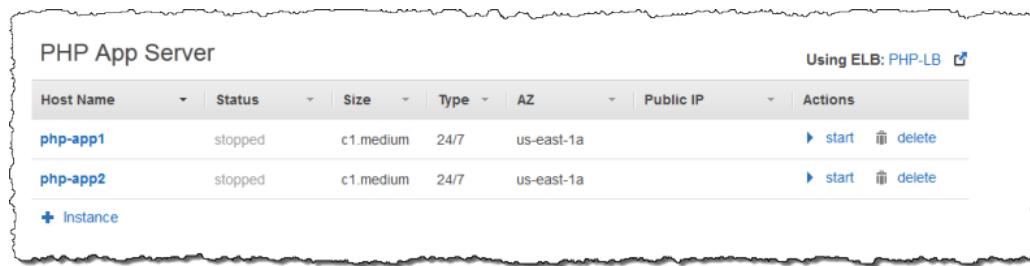
On the **Instances** page, click **Stop All Instances** and click **Stop** when asked confirm the operation.



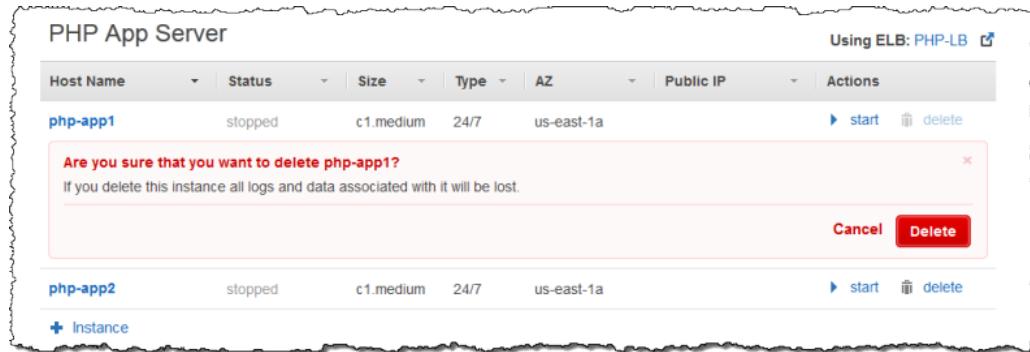
After you click **Stop**, AWS OpsWorks terminates the associated Amazon EC2 instances, but not any associated resources such as Elastic IP addresses or Amazon EBS volumes.

## 2. Delete all Instances

Stopping the instance just terminates the associated Amazon EC2 instances. After the instances status is in the stopped state, you must delete each instance. In the **PHP App Server** layer click **delete** in the php-app1 instance's **Actions** column.



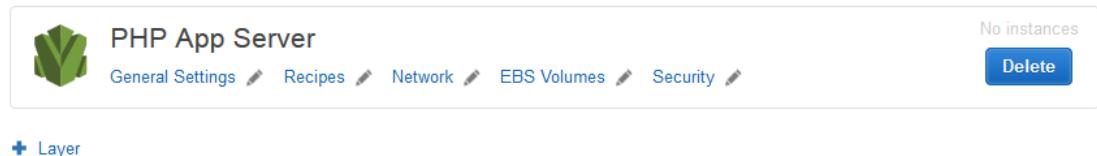
AWS OpsWorks then asks you to confirm the deletion, and shows you any dependent resources. You can choose to keep any or all of these resources. This example has no dependent resources, so just click **Delete**.



Repeat the process for php-app2 and the **MySQL** instance, db-master1. Notice that db-master1 has an associated Amazon Elastic Block Store volume, which is selected by default. Leave it selected to delete the volume along with the instance.

### 3. Delete the Layers.

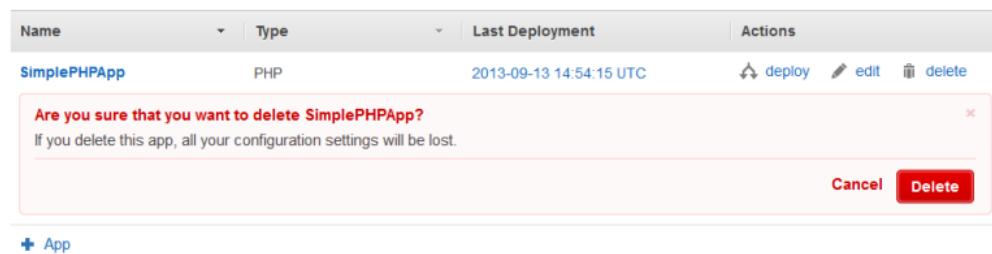
On the **Layers** page, click **Delete** and then click **Delete** to confirm.



Repeat the process for the **MySQL** layer.

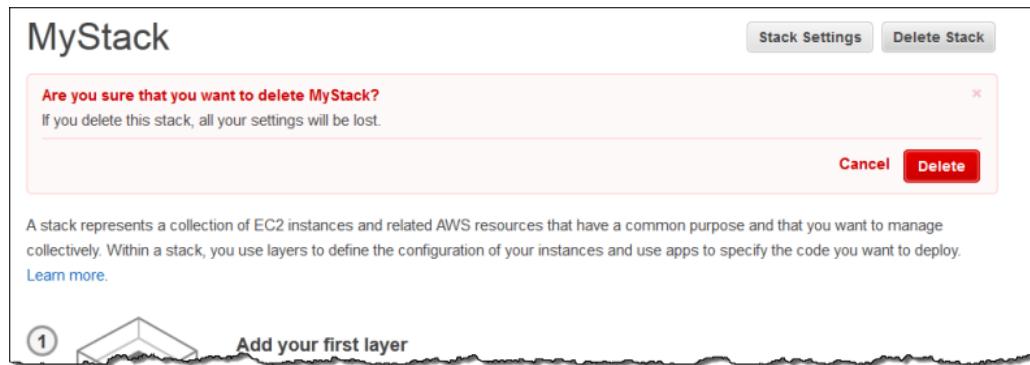
### 4. Delete the App

On the **Apps** page, click **delete** in the **SimplePHPApp** app's **Actions** column, and then click **Delete** to confirm.



### 5. Delete MyStack

On the **Stack** page, click **Delete Stack** and then click **Delete** to confirm.



# Stacks

---

The stack is the top-level AWS OpsWorks entity. It represents a set of instances that you want to manage collectively, typically because they have a common purpose such as serving PHP applications. In addition to serving as a container, a stack handles tasks that apply to the group of instances as a whole, such as managing applications and cookbooks.

For example, a stack whose purpose is to serve PHP applications might look something like the following:

- A set of PHP application server instances, each of which handles a portion of the incoming traffic.
- A load balancer instance, which takes incoming traffic and distributes it across the PHP application servers.
- A database instance, which serves as a back-end data store for the PHP application servers.

A common practice is to have multiple stacks that represent different environments. A typical set of stacks consists of:

- A development stack to be used by developers to add features, fix bugs, and perform other development and maintenance tasks.
- A staging stack to verify updates or fixes before exposing them publicly.
- A production stack, which is the public-facing version that handles incoming requests from users.

This section describes the basics of working with stacks.

## Topics

- [Create a New Stack \(p. 41\)](#)
- [Running a Stack in a VPC \(p. 44\)](#)
- [Update a Stack \(p. 50\)](#)
- [Clone a Stack \(p. 51\)](#)
- [Run Stack Commands \(p. 52\)](#)
- [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#)
- [Shut Down a Stack \(p. 55\)](#)

# Create a New Stack

## To create a stack

1. On the AWS OpsWorks dashboard, click **Add stack**.
2. On the **Add Stack** page, configure the stack settings:

### Name

(Required) A designation used to identify the stack in the AWS OpsWorks console.

### Region

(Required) The AWS region where the instances will be launched.

### VPC

(Optional) The ID of the VPC that the stack is to be launched into. All instances will be launched into this VPC, and you cannot change the ID later.

- If your account supports EC2 Classic, you can specify **No VPC** (the default value) if you don't want to use a VPC.
- If your account does not support EC2 Classic, you must use a VPC. However, you can specify **Default VPC**, which combines the ease of use of EC2 Classic with the benefits of VPC networking features. For more information, see [Your Default VPC and Subnets](#).

For more information on VPCs, see [Amazon Virtual Private Cloud](#). For more information on how to use AWS OpsWorks with a VPC, see [Running a Stack in a VPC \(p. 44\)](#).

### Default Availability Zone/Default Subnet

(Optional) This setting depends on whether you are creating your stack in a VPC:

- If your account supports EC2 Classic and you set **VPC to No VPC**, this setting is labeled **Default Availability Zone**, which specifies the default AWS Availability Zone where the instances will be launched.
- If your account does not support EC2 Classic or you choose to specify a VPC, this field is labeled **Default subnet**, which specifies the default subnet where the instances will be launched. You can launch an instance in other subnets by overriding this value when you create the instance. Each subnet is associated with one Availability Zone.

For more information on EC2 Classic, see [Supported Platforms](#). For more information on how to run a stack in a VPC, see [Running a Stack in a VPC \(p. 44\)](#).

### Default operating system

(Optional) The operating system that is installed by default on each instance. AWS OpsWorks supports three standard AMIs: [Amazon Linux](#), [Ubuntu 12.04 LTS](#), and [Ubuntu 14.04 LTS](#). The default option is Amazon Linux. If you select **Use custom AMI**, the default operating system is determined by a custom AMI, which you specify when you create instances. A custom AMI must be based on one of the standard AMIs. For more information, see [Using Custom AMIs \(p. 107\)](#).

### Default root device type

Determines the type of storage available to the instance. The default root device is the instance store, but you can also specify an Amazon EBS-backed root device. For more information, see [Storage](#).

### IAM Role

(Optional) The stack's AWS Identity and Access Management (IAM) role, which AWS OpsWorks uses to interact with AWS on your behalf. You should use a role that was created by AWS OpsWorks to ensure that it has all the required permissions. If you don't have an existing AWS OpsWorks role, select **New IAM Role** to have AWS OpsWorks create a new IAM role for you. For more information, see [Allowing AWS OpsWorks to Act on Your Behalf \(p. 293\)](#).

### Default SSH key

(Optional) The default Secure Shell (SSH) cryptographic key that instances will use to authenticate an SSH connection. The default value is none.

### Default IAM Instance Profile

(Optional) The default role to be associated with the stack's Amazon EC2 instances. If you want to specify permissions that are needed by apps running on EC2 instances (for example, if your apps access Amazon S3 buckets), you should choose an existing instance profile (role) that has the right permissions. Otherwise, you should select **New IAM Instance Profile** to let AWS OpsWorks create the instance profile for you. For more information, see [Specifying Permissions for Apps Running on EC2 instances \(p. 296\)](#).

### Host name theme

(Optional) A string that is used to generate a default hostname for each instance. The default value is **Layer Dependent**, which uses the short name of the instance's layer and appends a unique number to each instance. For example, the role-dependent **Load Balancer** theme root is "lb". The first instance you add to the layer is named "lb1", the second "lb2", and so on.

### Stack color

(Optional) The hue used to represent the stack on the AWS OpsWorks console. You can use different colors for different stacks to help distinguish, for example, among development, staging, and production stacks.

## Add Stack

Name	<input type="text" value="MyStack"/>
Region	<input type="text" value="US East (N. Virginia)"/>
VPC	<input type="text" value="No VPC"/>
Default Availability Zone	<input type="text" value="us-east-1a"/>
Default operating system <small>NEW</small>	<input type="text" value="Amazon Linux"/> <small>Need a different OS? <a href="#">Let us know.</a></small>
Default root device type	<input checked="" type="radio"/> Instance store <input type="radio"/> EBS backed
IAM role	<input type="text" value="New IAM role"/>
Default SSH key	<input type="text" value="Do not use a default SSH key"/>
Default IAM instance profile	<input type="text" value="aws-opsworks-ec2-role"/>
Hostname theme	<input type="text" value="Layer Dependent"/>
Stack color	<div style="background-color: #ccc; width: 100px; height: 15px; margin-bottom: 5px;"></div> <div style="display: flex; justify-content: space-around; align-items: center;"><span>#112233</span><span>#4472C4</span><span>#E64A19</span><span>#808080</span><span>#FFA500</span><span>#00FFFF</span><span>#008000</span><span>#800080</span><span>#FF0000</span><span>#00008B</span></div>
<hr/>	
Configuration Management	
Chef version	<input checked="" type="radio"/> 11.10 <small>NEW DEFAULT</small> <small>Need a different configuration management option? <a href="#">Let us know.</a></small>
	<input type="radio"/> 11.4
	<input type="radio"/> 0.9 <small>DEPRECATED</small>
Use custom Chef cookbooks	<input type="checkbox"/> No
Custom JSON	<input type="text" value="optional"/>
<small>Enter custom JSON that is passed to your Chef recipes for all instances in your stack. You can use this to override and customize built-in recipes or pass variables to your own recipes. <a href="#">Learn more.</a></small>	
<hr/>	
Security	
Use OpsWorks security groups	<input checked="" type="checkbox"/> Yes
<small><a href="#">Cancel</a> <a href="#">Add Stack</a></small>	

- To set the advanced options, click **Advanced >>** to display the **Configuration Management** and **Security** sections.

**Configuration Management** has the following options:

#### Chef version

(Optional) Lets you specify which Chef version to use. Chef 0.9 is supported primarily for compatibility with the initial OpsWorks release and is not recommended for new stacks. For more information on Chef versions, see [Chef Versions \(p. 161\)](#).

#### Use custom Chef Cookbooks

Selecting **Yes** lets you use custom Chef cookbooks to customize the behavior of your layers. The default setting is **No**. For more information on custom cookbooks, see [Cookbooks and Recipes \(p. 153\)](#).

If you select **Yes**, **Add Stack** displays several additional settings that provide AWS OpsWorks with the information it needs to deploy the custom cookbooks from their repository to the stack's instances. For more information, see [Installing Custom Cookbooks \(p. 171\)](#).

#### Custom Chef JSON

(Optional) Lets you specify custom JSON, which is incorporated into the stack configuration JSON object that is passed to instances and used by recipes. For more information, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#).

**Security** has one option, **Use OpsWorks security groups**, which allows you to specify whether to associate the AWS OpsWorks built-in security groups with the stack's layers.

AWS OpsWorks provides a standard set of built-in security groups—one for each layer—which are associated with layers by default. **Use OpsWorks security groups** allows you to instead provide your own custom security groups. For more information on security groups, see [Amazon EC2 Security Groups](#). **Use OpsWorks security groups** has the following settings:

- **Yes** - AWS OpsWorks automatically associates the appropriate built-in security group with each layer (default setting).

You can associate additional security groups with a layer after you create it but you cannot delete the built-in security group.

- **No** - AWS OpsWorks does not associate built-in security groups with layers.

You must create appropriate EC2 security groups and associate a security group with each layer that you create. However, you can still manually associate a built-in security group with a layer on creation; custom security groups are required only for those layers that need custom settings.

Note the following:

- If **Use OpsWorks security groups** is set to **Yes**, you cannot restrict a default security group's port access settings by adding a more restrictive security group to a layer. With multiple security groups, Amazon EC2 uses the most permissive settings. In addition, you cannot create more restrictive settings by modifying the built-in security group configuration. When you create a stack, AWS OpsWorks overwrites the built-in security groups' configurations, so any changes that you make will be lost the next time you create a stack. If a layer requires more restrictive security group settings than the built-in security group, set **Use OpsWorks security groups** to **No**, create custom security groups with your preferred settings, and assign them to the layers on creation.
- If you accidentally delete an AWS OpsWorks security group and want to recreate it, it must be an exact duplicate of the original, including the same capitalization for the group name. Instead of recreating the group manually, the preferred approach is to have AWS OpsWorks perform the task for you. Just create a new stack in the same AWS region—and VPC, if present—and AWS OpsWorks will automatically recreate all the built-in security groups, including the one that you deleted. You can then delete the stack if you don't have any further use for it; the security groups will remain.

4. When you have finished configuring the stack, click **Add stack**.

**Note**

You can override most default settings when you create instances.

## Running a Stack in a VPC

You can control user access to a stack's instances by creating it in a virtual private cloud (VPC). For example, you might not want users to have direct access to your stack's app servers or databases and instead require that all public traffic be channeled through an Elastic Load Balancer.

The basic procedure for running a stack in a VPC is:

1. Create an appropriately configured VPC, by using the Amazon VPC console or API, or an AWS CloudFormation template.
2. Specify the VPC ID when you create the stack.
3. Launch the stack's instances in the appropriate subnet.

The following section briefly describes how VPCs work, and the remaining sections describe a simple example: how to run a PHP app server stack in a VPC, using the VPC configuration that was described earlier in this section.

**Topics**

- [VPC Basics \(p. 44\)](#)
- [Create a VPC for an AWS OpsWorks Stack \(p. 46\)](#)
- [Create a Stack in the VPC \(p. 48\)](#)

## VPC Basics

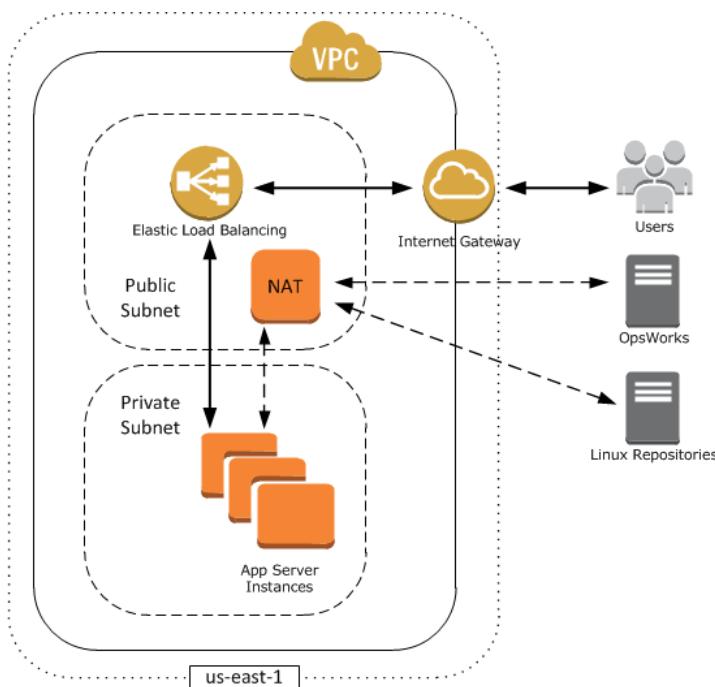
For a detailed discussion of VPCs, see [Amazon Virtual Private Cloud](#). Briefly, a VPC consists of one or more *subnets*, each of which contains one or more instances. Each subnet has an associated routing table that directs outbound traffic based on its destination IP address.

- Instances within a VPC can generally communicate with each other, regardless of their subnet.
- Subnets whose instances can communicate with the Internet are referred to as *public subnets*.
- Subnets whose instances can communicate only with other instances in the VPC and cannot communicate directly with the Internet are referred to as *private subnets*.

AWS OpsWorks requires the VPC to be configured so that every instance in the stack, including instances in private subnets, has access to the following endpoints:

- The AWS OpsWorks service, <https://opsworks-instance-service.us-east-1.amazonaws.com>.
- Amazon S3
- The package repositories for Amazon Linux or Ubuntu Linux, depending on which operating system you specify.
- Your app and custom cookbook repositories.

There are a variety of ways to configure a VPC to provide this connectivity. The following is a simple example of how you could configure a VPC for an AWS OpsWorks app server stack.



This VPC has several components:

### Subnets

The VPC has two subnets, one public and one private.

- The public subnet contains a load balancer and a network address translation (NAT) instance, which can communicate with external addresses and with the instances in the private subnet.
- The private subnet contains the application servers, which can communicate with the NAT and load balancer in the public subnet but cannot communicate directly with external addresses.

### Internet gateway

The Internet gateway allows instances with public IP addresses, such as the load balancer, to communicate with addresses outside the VPC.

### Load balancer

The Elastic Load Balancing load balancer takes incoming traffic from users, distributes it to the app servers in the private subnet, and returns the responses to users.

### NAT

The (NAT) instance provides the app servers with limited Internet access, which is typically used for purposes such as downloading software updates from an external repository. All AWS OpsWorks instances must be able to communicate with AWS OpsWorks and with the appropriate Linux repositories. One way to handle this issue is to put a NAT instance with an associated Elastic IP address in a public subnet. You can then route outbound traffic from instances in the private subnet through the NAT.

#### Tip

A single NAT instance creates a single point of failure in your private subnet's outbound traffic. You can improve reliability by configuring the VPC with a pair of NAT instances that take over for each other if one fails. For more information, see [High Availability for Amazon VPC NAT Instances](#).

The optimal VPC configuration depends on your AWS OpsWorks stack. The following are a few examples of when you might use certain VPC configurations. For examples of other VPC scenarios, see [Scenarios for Using Amazon VPC](#).

### Working with one instance in a public subnet

If you have a single-instance stack with no associated private resources—such as an Amazon RDS instance that should not be publicly accessible—you can create a VPC with one public subnet and put the instance in that subnet. If you are not using a default VPC, you must have the instance's layer assign an Elastic IP address to the instance. For more information, see [OpsWorks Layer Basics \(p. 58\)](#).

### Working with private resources

If you have resources that should not be publicly accessible, you can create a VPC with one public subnet and one private subnet. For example, in a load-balanced autoscaling environment, you can put all the Amazon EC2 instances in the private subnet and the load balancer in a public subnet. That way the Amazon EC2 instances cannot be directly accessed from the Internet; all incoming traffic must be routed through the load balancer.

The private subnet isolates the instances from Amazon EC2 direct user access, but they must still send outbound requests to AWS and the appropriate Linux package repositories. To allow such requests you can, for example, use a network address translation (NAT) instance with its own Elastic IP address and then route the instances' outbound traffic through the NAT. You can put the NAT in the same public subnet as the load balancer, as shown in the preceding example.

- If you are using a back-end database such as an Amazon RDS instance, you can put those instances in the private subnet. For Amazon RDS instances, you must specify at least two different subnets in different Availability Zones.
- If you require direct access to instances in a private subnet—for example, you want to use SSH to log in to an instance—you can put a bastion host in the public subnet that proxies requests from the Internet.

### Extending your own network into AWS

If you want to extend your own network into the cloud and also directly access the Internet from your VPC, you can create a VPN gateway. For more information, see [Scenario 3: VPC with Public and Private Subnets and Hardware VPN Access](#).

## Create a VPC for an AWS OpsWorks Stack

This section shows how to create a VPC for an AWS OpsWorks stack by using an example [AWS CloudFormation](#) template. You can download the template from <http://cloudformation-templates-us-east-1.s3.amazonaws.com/OpsWorksinVPC.template>. For more information on how to manually create a VPC like the one discussed in this topic, see [Scenario 2: VPC with Public and Private Subnets](#). For details on how to configure routing tables, security groups, and so on, see the example template.

#### Tip

By default, AWS OpsWorks displays subnet names by concatenating their CIDR range and Availability Zone, such as 10.0.0.1/24 – us-east-1b. To make the names more readable, create a tag for each subnet with **Key** set to **Name** and **Value** set to the subnet name. AWS OpsWorks then appends the subnet name to the default name. For example, the private subnet in the following example has a tag with **Name** set to **Private**, which OpsWorks displays as 10.0.0.1/24 us-east - 1b - Private.

You can launch a VPC template using the AWS CloudFormation console with just a few steps. The following procedure uses the example template to create a VPC in US East (Northern Virginia) Region. For directions on how to use the template to create a VPC in other regions, see the [note \(p. 47\)](#) that follows the procedure.

#### To create the VPC

1. Open the [AWS CloudFormation console](#), select the **US East (N. Virginia)** region, and click **Create Stack**.

2. On the **Create Template** page, add a template name, select **Provide a Template URL** and enter the template URL: `http://cloudformation-templates-us-east-1.s3.amazonaws.com/OpsWorksInVPC.template`. Click **Continue**.

The screenshot shows the 'Create Stack' wizard with the 'SELECT TEMPLATE' tab selected. The 'Stack Name' field contains 'OpsWorks-in-VPC'. Under 'Template', the 'Provide a Template URL' option is selected, and the URL 'http://cloudformation-templates-us-east-1.s3.amazonaws.com/OpsWorksInVPC.template' is entered. A 'Continue' button is at the bottom right.

- You can also launch this stack by going to [AWS CloudFormation Sample Templates](#), locating the AWS OpsWorks VPC template, and clicking **Launch Stack**.
3. On the **Specify Parameters** page, accept the default values and click **Continue**.
  4. On the **Add Tags** page, create a tag with **Key** set to **Name** and **Value** set to the VPC name. This tag will make it easier to identify your VPC when you create an AWS OpsWorks stack.
  5. Click **Continue** and then **Close** to launch the stack.

**Note:** You can create the VPC in other regions by using either of the following approaches:

- Go to [Using Templates in Different Regions](#), click the appropriate region, locate the AWS OpsWorks VPC template, and click **Launch Stack**.
- Copy the template file to your system, select the appropriate region in the [AWS CloudFormation console](#), and use the **Create Stack** wizard's **Upload a Template File** option to upload the template from your system.

The example template includes outputs that provide the VPC, subnet, and load balancer IDs that you will need to create the AWS OpsWorks stack. You can see them by clicking the **Outputs** tab at the bottom of the AWS CloudFormation console window.

Stack: OpsWorks-in-VPC		
Description	Outputs	Resources
Stack Outputs		
<i>Output values may have been specified by the template author and will be available when stack creation is complete.</i>		
Key	Value	Description
VPC	vpc-d1ed9ebf	VPC
PublicSubnets	subnet-66ec9f08	Public Subnet
PrivateSubnets	subnet-6dec9f03	Private Subnet
LoadBalancer	OpsWorks-ElasticL-K0FQ6TD5S14A	Load Balancer

## Create a Stack in the VPC

Now that you've created a VPC, you can run an OpsWorks stack in it.

### To run a PHP app server stack in a VPC

1. Open the AWS OpsWorks console and click **Add Stack**.
2. Create a stack.
  - Set **Region** to the region that you created the VPC in. For this example, click **US East (N. Virginia)**.
  - Specify the VPC that you created in the previous section. To make VPCs easier to identify, AWS OpsWorks appends the Name tag that you specified to the VPC ID.
  - Set **Default Subnet** to the private subnet, which has a CIDR address range of 10.0.1.0/24.

## Add Stack

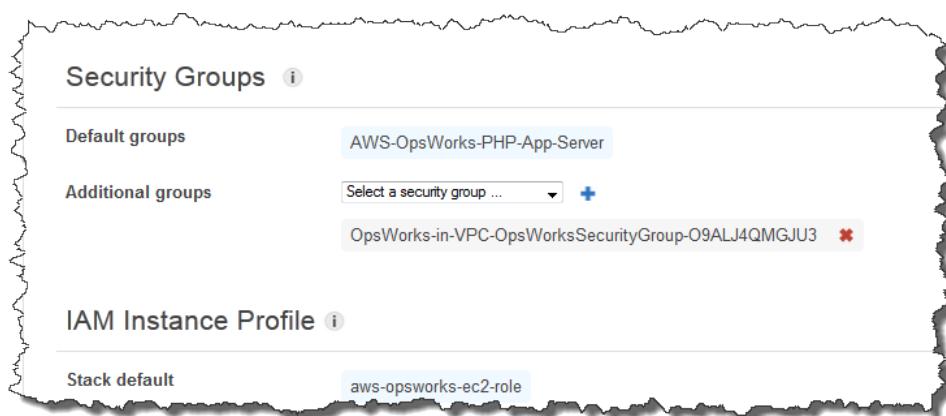
Name	<input type="text" value="VPCStack"/>
Region	<input type="text" value="US East (N. Virginia)"/>
VPC <small>NEW</small>	<input type="text" value="vpc-d1ed9ebf - OpsWorks-VPC"/>
Default subnet <small>NEW</small>	<input type="text" value="10.0.1.0/24 - us-east-1b - Private"/>
Default operating system	<input type="text" value="Amazon Linux"/>
Default root device type	<input checked="" type="radio"/> Instance store <input type="radio"/> EBS backed
IAM role	<input type="text" value="aws-opsworks-service-role-alpha"/>
Default SSH key	<input type="text" value="Do not use a default SSH key"/>
Default IAM instance profile	<input type="text" value="aws-opsworks-ec2-role"/>
Host name theme	<input type="text" value="Layer Dependent"/>
Stack color	<div style="background-color: #333; width: 15px; height: 10px;"></div> <div style="background-color: #666; width: 15px; height: 10px;"></div> <div style="background-color: #0070C0; width: 15px; height: 10px;"></div> <div style="background-color: #00AEEA; width: 15px; height: 10px;"></div> <div style="background-color: #2ECC71; width: 15px; height: 10px;"></div> <div style="background-color: #E67E22; width: 15px; height: 10px;"></div> <div style="background-color: #D35400; width: 15px; height: 10px;"></div>
<a href="#">Advanced <small>NEW »</small></a>	

3. Add a PHP App Server layer to the stack and attach the VPC's load balancer.

## Add Layer

Layer type	PHP App Server
The PHP Application Server layer is a blueprint for instances that function as PHP application servers. By default PHP 5.3 and Apache 2.2 are installed. <a href="#">Learn more</a> .	
Elastic Load Balancer	OpsWorks-ElasticL-K0FQ6TD5S14A
<a href="#">Cancel</a> <a href="#" style="background-color: #0070C0; color: white; border-radius: 5px; padding: 2px 10px;">Add Layer</a>	

4. Add a security group to the PHP App Server layer to provide the necessary ingress and egress rules for the VPC to function correctly. The security group name will start with *CFN-Stack-NameOpsWorksSecurityGroup*, followed by an alphanumeric code. For this example, the name is *OpsWorks-in-VPC-OpsWorksSecurityGroup-O9ALJ4QMGJU3*.
  1. On the **Layers** page, for PHP App Server, click **Security** and then click **Edit**.
  2. Under **Security Groups**, select the security group from the **Additional groups** list and click **+** to add it to the layer's security groups.
  3. Click **Save** to update the layer's configuration.



5. Add one or more instances to the PHP App Server layer with the default settings. You can explicitly specify an instance's subnet, but for this example just accept the default private subnet that you specified when you created the stack.
6. Add the SimplePHP app to the stack. On the Apps page, click **Add an app** or **+ App** and do the following:
  - Set **Name** to SimplePHPApp.
  - Set **Repository type** to Git.
  - Set **Repository URL** to `git://github.com/amazonwebservices/opsworks-demo-php-simple-app.git`.
  - Set **Branch/Revision** to `version1`.

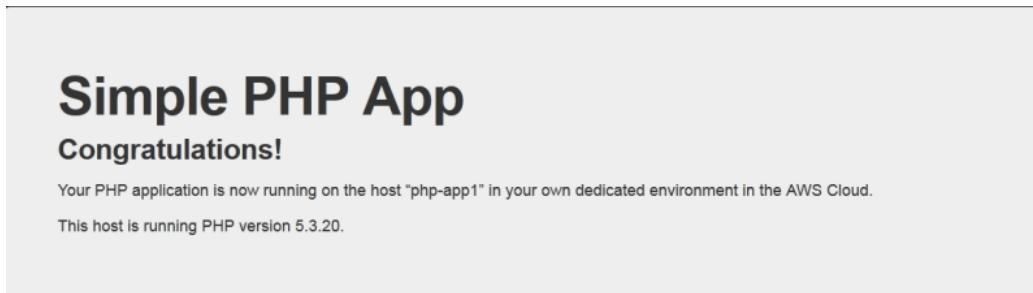
Accept defaults for the other settings and click **Add App**. For a more detailed description, see [Step 4: Create and Deploy an App \(p. 16\)](#)

7. On the **Instances** page, start the PHP App Server instance, which will also deploy SimplePHP to the instance.

**Tip**

If an instance status does not progress beyond **booting**, the VPC's network or security group settings might not be configured correctly. For guidance on how to configure a VPC, see [Scenario 2: VPC with Public and Private Subnets](#).

8. On the **Layers** page, click the load balancer's ID in the PHP App Server layer's **Elastic Load Balancer** column.
9. On the load balancer's details page, after the health check is green, click the DNS name to run SimplePHP, which displays the following.



## Update a Stack

After you have created a stack, you can update the configuration at any time.

### To edit a stack

1. On the Stack page, click **Stack Settings**, and then click **Edit**.
2. On the **Settings** page, make the changes that you want and click **Save**. The settings are the same as those discussed in [Create a New Stack \(p. 41\)](#). Refer to that topic for details. However, note the following:
  - You can modify any of the settings except the region and the VPC ID.
  - If your stack is running in a VPC, the settings include a **Default subnet** setting, which lists the VPC's subnets. If your stack is not running in a VPC, the setting is labeled **Default Availability Zones**, which lists the region's Availability Zones.
  - If you change any of the default instance settings, such as **Hostname theme** or **Default SSH key**, the new values apply only to any new instances you create, not to existing instances.
  - Changing the **Name** changes the name that is displayed by the console; it does not change the underlying short name that AWS OpsWorks uses to identify the stack.
  - Before you change **Use OpsWorks security groups** from **Yes** to **No**, each layer must have at least one security group in addition to the layer's built-in security group. For more information, see [How to Edit an OpsWorks Layer \(p. 59\)](#).

AWS OpsWorks then deletes the built-in security groups from every layer.

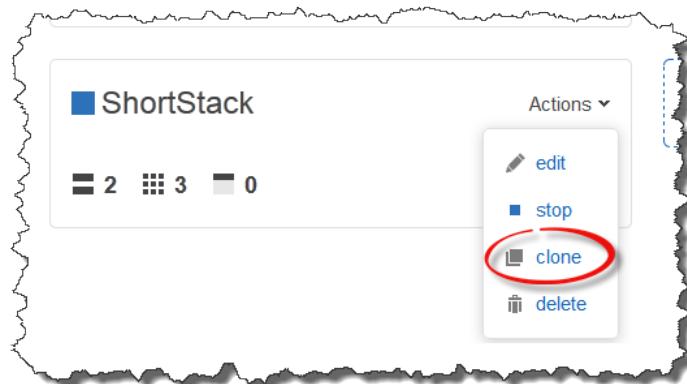
- If you change **Use OpsWorks security groups** from **No** to **Yes**, AWS OpsWorks adds the appropriate built-in security group to each layer but does not delete the existing security groups.

# Clone a Stack

It is sometimes useful to create multiple copies of a stack. For example, you might want to run the same stack in multiple AWS regions, or you might use an existing stack as a starting point for a new stack. The simplest approach is to clone the source stack.

## To clone a stack

1. On the AWS OpsWorks dashboard, in the box for the stack that you want to clone, click **Actions**, and then select **clone**.



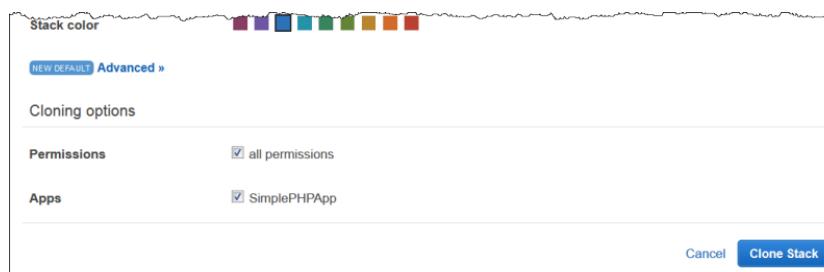
2. On the **Clone stack** page, modify any configuration settings that you want. Initially, the settings for the cloned stack are identical to those for the source stack except that the word "copy" is appended to the stack name. For information about these settings, see [Create a New Stack \(p. 41\)](#). There are also two additional optional settings:

### Permissions

If the **all permissions** check box is selected (the default), the source stack permissions are applied to the cloned stack.

### Apps

Lists apps that are deployed on the source stack. For each app listed, if the corresponding check box is selected (the default), the app will be deployed to the cloned stack.



Permissions	<input checked="" type="checkbox"/> all permissions
Apps	<input checked="" type="checkbox"/> SimplePHPApp

3. When all the settings are as you want them, click **Clone stack**.

When you click **Clone stack**, AWS OpsWorks creates a new stack that consists of the source stack's layers and optionally its apps and permissions. The layers have the same configuration as the originals, subject to any modifications that you made. However, cloning does not create any instances. You must add an appropriate set of instances to each layer of the cloned stack and then start them. As with any stack, you can perform normal management tasks on a cloned stack, such as adding, deleting, or modifying layers or adding and deploying apps.

To make the cloned stack operational, start the instances. AWS OpsWorks sets up and configures each instance according to its layer membership. It also deploys any applications, just as it does with a new stack.

## Run Stack Commands

AWS OpsWorks supports several commands that you can use on a stack to perform the following operations:

### Install Dependencies

Installs all packages.

### Update Dependencies

Updates all packages.

### Update Custom Cookbooks

Deploys an updated set of custom Chef cookbooks from the repository to each instance's cookbooks cache.

### Execute Recipes

Executes a specified set of recipes.

### Setup

Runs the Setup recipes.

### Configure

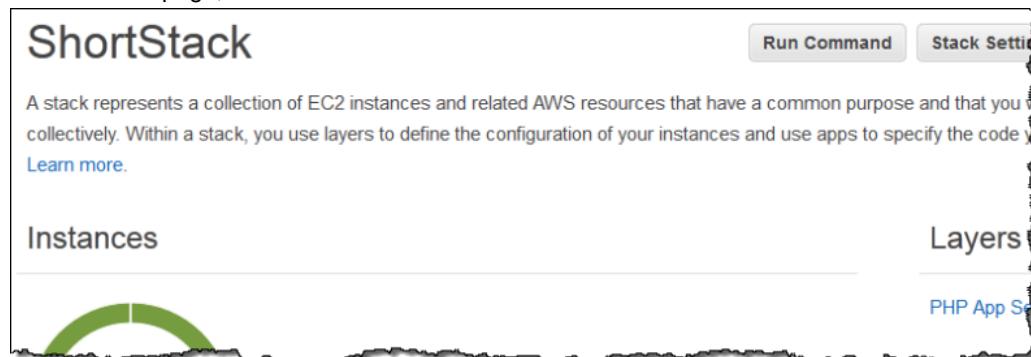
Runs the Configure recipes.

### Important

After running **Update Dependencies** to do a full operating system upgrade, for example from Amazon Linux 2013.09 to 2014.03, we recommend that you also run **Setup**. This ensures that services are correctly restarted.

### To run a stack command

1. On the **Stack** page, click **Run Command**.



2. On the **Run Command** page, under **Settings**, click the **Command** drop-down list and select the command that you want to run.

## Run Command

Settings

Command    
Update Dependencies  
Install Dependencies  
Update Custom Cookbooks  
Execute Recipes  
Setup  
Configure

[Advanced »](#)

Instances i

OpsWorks will run this command on **3 of 3** instances. The assigned recipes are run on all selected instances.

**PHP App Server**  php-app1 ●  php-app2 ●  
Click to select instances in this layer

**MySQL**  db-master1 ●  
Click to select instances in this layer

[Cancel](#)

You can specify the following settings:

**Comment**

(Optional) Any custom remarks you care to add.

**Recipes to execute**

(Required) This field appears if you select the **Execute Recipes** command. Enter the recipes to be executed using the standard `cookbook::recipe` format, separated by commas.

**Custom Chef JSON**

(Optional) A custom JSON object, to be incorporated into the stack configuration JSON. For more information, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#).

**Instances**

(Optional) The instances on which to execute the command. All online instances are selected by default. Select the appropriate layers or instances to run the command on a subset of instances.

3. When the command is configured, click the button at the lower right, which will be labeled with the command name, to run the command on the specified instances.

**Note**

You might see `execute_recipes` executions that you did not run listed on the Deployment and Commands page. This is usually the result of a permissions change, such as granting or removing SSH permissions for a user. When you make such a change, AWS OpsWorks uses `execute_recipes` to update permissions on the instances.

## Use Custom JSON to Modify the Stack Configuration JSON

For several AWS OpsWorks actions, you can specify custom JSON, which is incorporated into the stack configuration JSON object that is passed to instances and used by recipes. The most common use of custom JSON is to override the default AWS OpsWorks configuration settings in the stack configuration

## AWS OpsWorks User Guide

### Use Custom JSON to Modify the Stack Configuration

#### JSON

JSON or the built-in cookbooks with stack-specific values. You can also use custom JSON to add custom elements to the stack configuration JSON, which can then be accessed by your custom recipes. For information, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

You use custom JSON in one of two ways:

- When you create, update, or clone a stack.

The custom JSON is incorporated into the stack configuration JSON and passed to instances for all subsequent events.

- When you run a deployment or stack command.

The custom JSON is incorporated into the stack configuration JSON and passed to instances only for that event.

A custom JSON object must be valid JSON and formatted as follows:

```
{  
  "att1": "value1",  
  "att2": "value2"  
  ...  
}
```

For example, suppose that you want to change some default Apache configuration settings. They are defined in the built-in apache2 cookbook's apache.rb file, which you can view at <https://github.com/aws/opsworks-cookbooks>. Simply take the following steps:

1. On the stack page, click **Stack Settings** and then click **Edit**.
2. In the **Custom Chef JSON** box, add an appropriate object. In this example, the object modifies two Apache configuration settings, the keep-alive timeout and the log rotate schedule.



For all subsequent lifecycle events, AWS OpsWorks passes the instance's updated stack configuration JSON with the custom Apache configuration values overriding the defaults. Recipes can retrieve custom values by using standard Chef node syntax, which maps directly to the hierarchy in the JSON object. For example, recipes can retrieve the new keepalivetimeout value by using the following:

```
node[:apache][:keepalivetimeout]
```

If you want to do string interpolation, enclose the node value in curly brackets, as follows.

```
#{{node[:apache][:keepalivetimeout]}}
```

Custom JSON is not limited to overriding standard values; you can also specify values for arbitrary JSON attributes. This approach can be a useful way to pass data to your custom recipes. AWS OpsWorks adds those attributes to the stack configuration JSON, and your custom recipes can retrieve the values by using standard Chef syntax.

**Note**

Custom JSON is limited to 20 KB. If you need more capacity, we recommend storing some of the data on Amazon S3. Your custom recipes can then use the AWS CLI or Ruby SDK to download the data from the bucket to your instance. You will also need to modify the instance profile to allow CLI commands or applications running on the instance to access the bucket.

## Shut Down a Stack

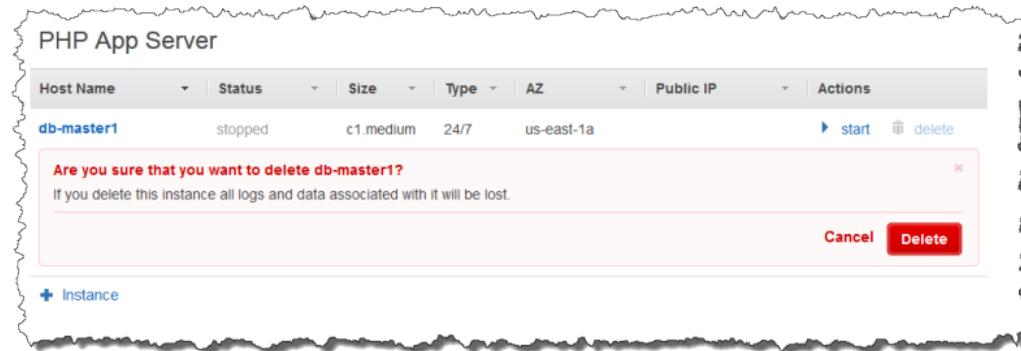
If you no longer need a stack, you can shut it down.

### To shut down a stack

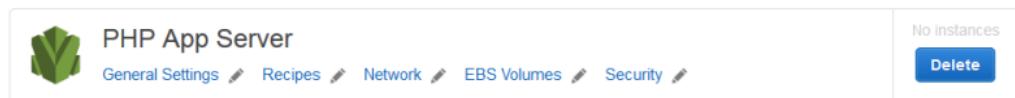
1. On the AWS OpsWorks dashboard, click the stack that you want to shut down.
2. In the navigation pane, click **Instances**.
3. On the **Instances** page, click **Stop all Instances**.



4. After the instances have stopped, for each instance in the layer, click **delete** in the **Actions** column. When prompted to confirm, click **Yes, Delete**.



5. When all the instances are deleted, in the navigation pane, click **Layers**.
6. On the **Layers** page, for each layer in the stack, click **delete**. When a confirmation prompt appears, click **Yes, Delete**.



+ Layer

7. When all the layers are deleted, in the navigation pane, click **Apps**.
8. On the **Apps** page, for each app in the stack, click **delete** in the **Actions** column. When prompted to confirm, click **Yes, Delete**.

## Apps

An app represents code stored in a repository that you want to install on application server instances. When you deploy the app, OpsWorks downloads the code from the repository to the specified server instances. [Learn more](#).

+ App

9. When all the apps are deleted, in the navigation pane, click **Stack**.
10. On the stack page, click **Delete stack**. When prompted to confirm, click **Yes, Delete**.

# Layers

---

Every stack contains one or more layers, each of which represents an application component, such as a load balancer or an application server. AWS OpsWorks supports two types of layer:

- An OpsWorks layer is essentially a blueprint for a set of one or more Amazon EC2 instances. The layer defines which packages and applications are installed, how they are configured, and so on. For example, each instance that belongs to a PHP App Server layer has an [Apache2](#) server with mod\_php installed and usually at least one PHP application.
- A service layer represents an AWS service, such as an Amazon RDS instance. OpsWorks currently offers one service layer, Amazon RDS.

This section is primarily concerned with OpsWorks layers; a service layer works somewhat differently, and each has its own set of procedures. For more information see [Amazon RDS Service Layer \(p. 77\)](#).

As you work with OpsWorks layers, keep the following in mind:

- Each layer in a stack must have at least one instance and can optionally have multiple instances. For example, a PHP App Server layer typically has multiple instances but a Ganglia layer typically has only one.
- Each instance in a stack must be a member of at least one layer.

You cannot configure an instance directly, except for some basic settings such as the SSH key and hostname. You must create and configure an appropriate layer, and add the instance to the layer.

## Note

You cannot add Amazon EC2 instances to a service layer. You must use the service's console, API, or CLI to create an instance of the service. You then use a service layer to incorporate the instance into your stack.

Amazon EC2 instances can optionally be a member of multiple OpsWorks layers. In that case, AWS OpsWorks runs the recipes to install and configure packages, deploy applications, and so on for each of the instance's layers. However, an instance's layers must be compatible with each other. For example, the HAProxy layer is not compatible with the Static Web Server layer because they both bind to port 80. An instance can be a member of only one of those layers. For more information, see [Appendix A: AWS OpsWorks Layer Reference \(p. 354\)](#).

By assigning an instance to multiple layers, you could, for example do the following:

- Reduce expenses by hosting the database server and the load balancer on a single instance.  
Assign an instance to both the HAProxy and MySQL layers.
- Use one of your application servers for administration.

Create a custom administrative layer and add one of the application server instances to that layer. The administrative layer's recipes configure that application server instance to perform administrative tasks, and install any additional required software. The other application server instances are just application servers.

AWS OpsWorks includes a set of standard layers that address common use cases such as serving PHP applications or static HTML pages. You can use these layers as they are or [customize them \(p. 230\)](#) by implementing custom Chef recipes and assigning them to the appropriate lifecycle events. If none of the standard layers meet your requirements, even with customization, AWS OpsWorks includes a custom layer, which you can use to handle a wide variety of scenarios. Each stack can have at most one of each standard layer, but you can implement any number of custom layers.

This section describes how to create each AWS OpsWorks layer. For more information about available layers, including standard recipes, compatibility with other layers, and so on, see [Appendix A: AWS OpsWorks Layer Reference \(p. 354\)](#).

#### Topics

- [OpsWorks Layer Basics \(p. 58\)](#)
- [Load Balancer Layers \(p. 69\)](#)
- [Database Layers \(p. 77\)](#)
- [Application Server Layers \(p. 81\)](#)
- [Custom AWS OpsWorks Layers \(p. 96\)](#)
- [Other Layers \(p. 98\)](#)

## OpsWorks Layer Basics

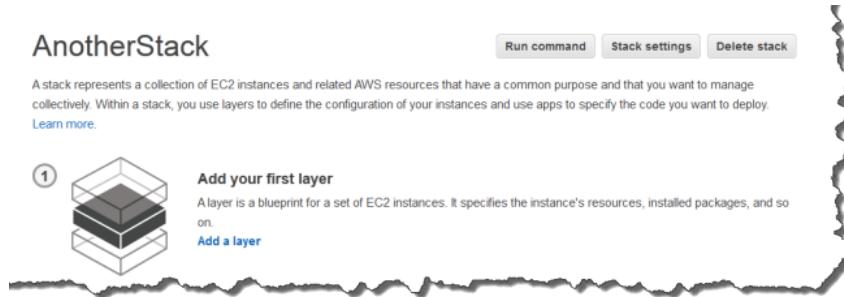
This section describes how to perform operations that are common to all OpsWorks layers.

#### Topics

- [How to Create an OpsWorks Layer \(p. 58\)](#)
- [How to Edit an OpsWorks Layer \(p. 59\)](#)
- [How to Use Auto Healing to Replace a Layer's Failed Instances \(p. 67\)](#)
- [How to Delete an OpsWorks Layer \(p. 68\)](#)

## How to Create an OpsWorks Layer

When you create a new stack, you see the following page:



### To add the first OpsWorks layer

1. Click **Add a Layer**.
2. On the **Add Layer** page, select the appropriate layer, which displays the layer's configuration options.
3. Configure the layer appropriately and click **Add Layer** to add it to the stack. The following sections describe how to configure the various layers.
4. Add instances to the layer and start them.

#### Note

If an instance is a member of multiple layers, you must add it to all of them before you start the instance. You cannot add an online instance to a layer.

To add more layers, open the **Layers** page and click **+ Layer** to open the **Add Layer** page.

When you start an instance, AWS OpsWorks automatically runs default setup and configuration recipes for each of the instance's layers to install and configure the appropriate packages and deploy the appropriate applications. You can [customize a layer's setup and configuration process \(p. 230\)](#) in a variety of ways, such as by assigning custom recipes to the appropriate lifecycle events. AWS OpsWorks runs custom recipes after the standard recipes for each event. For more information, see [Cookbooks and Recipes \(p. 153\)](#).

The following layer-specific sections describe how handle Steps 2 and 3 for the various AWS OpsWorks layers. For more information how to add instances, see [Adding an Instance to a Layer \(p. 103\)](#).

## How to Edit an OpsWorks Layer

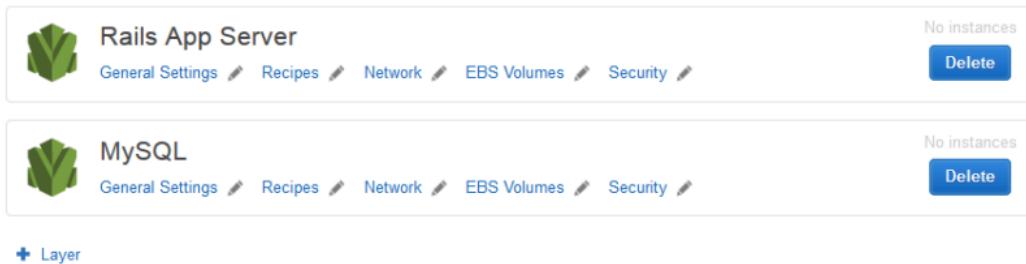
After you create a layer, some properties (such as AWS region) are immutable, but you can change most of the layer configuration at any time. Editing the layer also provides access to configuration settings that are not available when you create the layer.

### To edit an OpsWorks layer

1. In the navigation pane, click **Layers**.
2. On the **Layers** page, click a layer name to open the details page, which shows the current configuration.

## Layers

A layer is a blueprint for a set of EC2 instances. It specifies the instance's settings, resources, installed packages, profiles, and security groups. OpsWorks provides a set of layers for standard use cases such as application servers, which you can customize as needed. You can also create a custom layer whose configuration you can define from the ground up. [Learn more](#).



You can also click one of the links under the layer name to go directly to the associated tab.

3. Click **Edit** and then select the appropriate tab: **General Settings**, **Recipes**, **Network**, **EBS Volumes**, or **Security**. The settings on each tab are discussed in the following sections.

### Tip

Clicking the icon to the right of a tab name on the **Layers** page takes you directly to the associated tab's edit page.

The following sections describe the settings on the various tabs that are available to all layers. Some layers have additional layer-specific settings that you can edit; these appear at the beginning of the page. For example, the MySQL **Layer** page lets you modify the MySQL root password and specify whether to include the password in the stack configuration JSON for all of the stack's instances.

### Topics

- [General Settings \(p. 60\)](#)
- [Recipes \(p. 61\)](#)
- [Network \(p. 62\)](#)
- [EBS Volumes \(p. 63\)](#)
- [Security \(p. 65\)](#)

## General Settings

All layers have an **Auto healing enabled** setting, with which you can enable or disable auto healing for the layer's instances. The setting becomes effective as soon as you save the new configuration.

The remaining settings on this tab vary with the type of layer and are identical to the settings on the layer's **Add Layer** page. See the appropriate layer documentation for details. For example, the following shows the **General Settings** tab for a Rails App Server layer.

## Layer Rails App Server

General Settings   Recipes   Network   EBS Volumes   Security

Settings

Ruby version	2.0.0
Rails stack	<input checked="" type="radio"/> Apache2 and Passenger <input type="radio"/> nginx and Unicorn
Passenger version	4.0.39
RubyGems version	2.1.7
Install and manage Bundler	<input checked="" type="checkbox"/> Yes
Bundler version	1.5.1

Auto Healing i

Auto healing enabled	<input checked="" type="checkbox"/>
----------------------	-------------------------------------

Cancel Save

## Recipes

The **Recipes** tab includes the following settings.

### Built-in Chef Recipes

You can view the layer's built-in recipes but you cannot modify them.

### Custom Chef recipes

You can assign custom Chef recipes to the layer's lifecycle events. For more information, see [Executing Recipes \(p. 175\)](#).

### OS Packages

You can specify additional OS packages to be installed on new instances. To install new packages on existing instances, run the [Install Dependencies command \(p. 52\)](#).

## Layer Rails App Server

The screenshot shows the AWS OpsWorks 'Edit Layer' interface for a 'Rails App Server' layer. The 'Recipes' tab is active. Under 'Built-in Chef Recipes', it lists 22 recipes across four categories: Setup (12), Configure (5), Deploy (2), and Undeploy (1). Under 'Custom Chef Recipes', it says 'If you want to use Custom Chef recipes you need to [configure cookbooks](#) first.' In the 'OS Packages' section, there's a text input field 'Enter name' and a '+' button. At the bottom right are 'Cancel' and 'Save' buttons.

## Network

The **Network** tab includes the following settings.

### Elastic Load Balancing

You can apply or remove an Elastic Load Balancing load balancer.

### Automatically Assign IP Addresses

You can control whether AWS OpsWorks automatically assigns public or Elastic IP addresses to the layer's instances. Here's what happens when you enable this option:

- For instance store-backed instances, AWS OpsWorks automatically assigns an address each time the instance is started.
- For Amazon EBS-backed instances, AWS OpsWorks automatically assigns an address when the instance is started for the first time.
- If an instance belongs to more than one layer, AWS OpsWorks automatically assigns an address if you have enabled automatic assignment for at least one of the layers,

If your stack is running in a VPC, you have separate settings for public and Elastic IP addresses. The following table explains how these interact:

Public IP addresses		
Elastic IP addresses		
	Yes	No
Yes	Instances receive an Elastic IP address when they are started for the first time, or a public IP address if an Elastic IP cannot be assigned.	Instances receive an Elastic IP address when they are started for the first time.
No	Instances receive a public IP address each time they are started.	Instances receive only a private IP address, which is not accessible from outside the VPC.

**Note**

Instances must have a way to communicate with the AWS OpsWorks service, Linux package repositories, and cookbook repositories. If you specify no public or Elastic IP address, your VPC must include a component such as a NAT that allows the layer's instances to communicate with external sites. For more information, see [Running a Stack in a VPC \(p. 44\)](#).

If your stack is not running in a VPC, **Elastic IP addresses** is your only setting:

- **Yes:** Instances receive an Elastic IP address when they are started for the first time, or a public IP address if an Elastic IP address cannot be assigned.
- **No:** Instances receive a public IP address each time they are started.

## Layer Rails App Server

General Settings   Recipes   Network   EBS Volumes   Security

Elastic Load Balancing ⓘ

Elastic Load Balancer   No ELBs have been created in us-east-1. To add an ELB go to the [EC2 console](#).

Automatically Assign IP Addresses ⓘ

Public IP addresses   yes

Elastic IP addresses    No

Instances receive a public IP address every time they are started.

Cancel   Save

## EBS Volumes

The **EBS Volumes** tab includes the following settings.

**EBS optimized instances**

Whether the layer's instances should be Amazon EBS-backed.

**Additional EBS Volumes**

You can add [Amazon Elastic Block Store volumes](#) to or remove them from the layer's instances. When you start an instance, AWS OpsWorks automatically creates the volumes and attaches them to the instances. You can use the **Resources** page to manage a stack's EBS volumes. For more information, see [Resource Management \(p. 218\)](#).

- **Mount point** – (Required) Specify the mount point or directory where the EBS volume will be mounted.
- **RAID level** – (Optional) To use a RAID array, specify the level.

The default RAID level is **None**, which specifies a single-disk volume. You can also specify a level 0, level 1, or level 10 RAID array. For more information about RAID arrays, see [RAID](#).

- **# Disks** – (Optional) If you specified a RAID array, the number of disks in the array.

Each RAID level has a default number of disks, but you can select a larger number from the list.

- **Size total (GiB)** – (Required) The volume's size, in GB.

For a RAID array, this setting specifies the total array size, not the size of each disk.

- **Volume Type** – (Optional) Specify whether to create a standard or PIOPS volume.

The default value is **Standard**.

- **IOPS per disk** – (Required for PIOPS volumes) If you specify a PIOPS volume, you must also specify the **IOPS per disk**, which cannot be more than 30 times larger than the volume's size in GB.

For example, a 100 GB PIOPS volume cannot exceed 3000 IOPS per disk.

## Layer Rails App Server

Mount point	RAID level	# Disks	Size total (GiB)	per disk (GiB)	Volume Type	IOPS per disk
/vol/mountpoint	None	1	100	100.00	standard	-

For provisioned IOPS volumes, you can specify the IOPS rate when you create the volume. The ratio of IOPS provisioned and the volume size requested can be a maximum of 30 (in other words, a volume with 3000 IOPS must be at least 100 GB).

Cancel Save

When you add volumes to or remove them from a layer, note the following:

- If you add a volume, every new instance gets the new volume, but AWS OpsWorks does not update the existing instances.
- If you remove a volume, it applies only to new instances; the existing instances retain their volumes.

## Specifying a Mount Point

You can specify any mount point that you prefer. However, be aware that some mount points are reserved for use by AWS OpsWorks or Amazon EC2 and should not be used for Amazon EBS volumes.

The following mount points are reserved for use by AWS OpsWorks.

- `/srv/www`
- `/var/log/apache2` (Ubuntu)
- `/var/log/httpd` (Amazon Linux)
- `/var/log/mysql`
- `/var/www`

When an instance boots or reboots, autofs (an automounting daemon) uses ephemeral device mount points such as `/media/ephemeral0` for bind mounts. This operation takes place before Amazon EBS volumes are mounted. To ensure that your Amazon EBS volume's mount point does not conflict with autofs, do not specify an ephemeral device mount point. The possible ephemeral device mount points depend on the particular instance type, and whether it is instance store-backed or Amazon EBS-backed. To avoid a conflict with autofs, do the following:

- Verify the ephemeral device mount points for the particular instance type and backing store that you want to use.
- Be aware that a mount point that works for an instance store–backed instance might conflict with autofs if you switch to an Amazon EBS–backed instance, or vice versa.

**Tip**

If you want to change the instance store block device mapping, you can create a custom AMI.

For more information, see [Amazon EC2 Instance Store](#). For more information about how to create a custom AMI for AWS OpsWorks, see [Using Custom AMIs \(p. 107\)](#).

The following is an example of how to use a custom recipe to ensure that a volume's mount point doesn't conflict with autofs. You can adapt it as needed for your particular use case.

**To avoid a conflicting mount point**

1. Assign an Amazon EBS volume to desired layer but use a mount point such as `/mnt/workspace` that will never conflict with autofs.
2. Implement the following custom recipe, which creates an application directory on the Amazon EBS volume and links to it from `/srv/www/`. For more information on how to implement custom recipes, see [Cookbooks and Recipes \(p. 153\)](#) and [Customizing AWS OpsWorks \(p. 230\)](#).

```
mount_point = node['ebs']['raids'][ '/dev/md0']['mount_point'] rescue nil

if mount_point
  node[:deploy].each do |application, deploy|
    directory "#{{mount_point}}/#{application}" do
      owner deploy[:user]
      group deploy[:group]
      mode 00770
      recursive true
    end

    link "/srv/www/#{application}" do
      to "#{{mount_point}}/#{application}"
    end
  end
end
```

3. Add a `depends 'deploy'` line to the custom cookbook's `metadata.rb` file.
4. [Assign this recipe to the layer's Setup event. \(p. 175\)](#)

## Security

The **Security** tab includes the following settings.

**Security Groups**

A layer must have at least one associated security group. You specify how to associate security groups when you [create \(p. 41\)](#) or [update \(p. 50\)](#) a stack. AWS OpsWorks provides a standard set of built-in security groups, one for each layer.

- The default option is to have AWS OpsWorks automatically associate the appropriate built-in security group with each layer, such as `AWS-OpsWorks-PHP-App-Server` for the PHP App Server layer.

- You can also choose to not automatically associate built-in security groups and instead associate a custom security group with each layer when you create the layer.

For more information on security groups, see [Amazon EC2 Security Groups](#).

After the layer has been created, you can use **Security Groups** to add more security groups to the layer by selecting them from the **Custom security groups** list. You can also delete existing security groups by clicking the **x**, as follows:

- If you chose to have AWS OpsWorks automatically associate built-in security groups, you can delete custom security groups that you added earlier by clicking the **x**, but you cannot delete the built-in group.
- If you chose to not automatically associate built-in security groups, you can delete any existing security groups, including the original one, as long as the layer retains at least one group.

Note the following:

- You cannot restrict a built-in security group's port access settings by adding a more restrictive security group. When there are multiple security groups, Amazon EC2 uses the most permissive settings.
- You should not modify a built-in security group's configuration. When you create a stack, AWS OpsWorks overwrites the built-in security groups' configurations, so any changes that you make will be lost the next time you create a stack.

If you discover that you need more restrictive security group settings for one or more layers take these steps:

1. Create custom security groups with appropriate settings and add them to the appropriate layers.

Every layer in your stack must have at least one security group in addition to the built-in group, even if only one layer requires custom settings.

2. [Edit the stack configuration \(p. 50\)](#) and switch the **Use OpsWorks security groups** setting to **No**.

AWS OpsWorks automatically removes the built-in security group from every layer.

For more information on security groups, see [Amazon EC2 Security Groups](#). For the built-in security groups for each layer, see [Appendix A: AWS OpsWorks Layer Reference \(p. 354\)](#).

#### IAM Instance Profile

You can change the IAM profile for the layer's instances. For more information, see [Specifying Permissions for Apps Running on EC2 instances \(p. 296\)](#).

## Layer Rails App Server

The screenshot shows the AWS OpsWorks console with the 'Security' tab selected. Under 'Security Groups', there is a 'Default groups' section containing 'AWS-OpsWorks-Rails-App-Server' and a 'Custom security groups' section with a dropdown menu labeled 'Select a security group'. Under 'IAM Instance Profile', there is a 'Stack default' section with 'aws-opsworks-ec2-role' and a 'Layer profile' section with a dropdown menu labeled 'Use default stack profile'. At the bottom right are 'Cancel' and 'Save' buttons.

## How to Use Auto Healing to Replace a Layer's Failed Instances

Every instance has an AWS OpsWorks agent that communicates regularly with the service. AWS OpsWorks uses that communication to monitor instance health. If an agent does not communicate with the service for more than a small amount of time (approximately five minutes), AWS OpsWorks considers the instance to have failed.

If a layer has auto healing enabled—the default setting—AWS OpsWorks automatically replaces the layer's failed instances in three steps:

1. Terminates the Amazon EC2 instance and verifies that it has shut down.
2. Launches a new Amazon EC2 instance with the same host name, configuration, and layer membership.
3. Reattaches any Amazon EBS volumes, including volumes that were attached after the instance was originally started.

### Note

An instance can be a member of multiple layers. If any of those layers has auto healing enabled, AWS OpsWorks heals the instance if it fails.

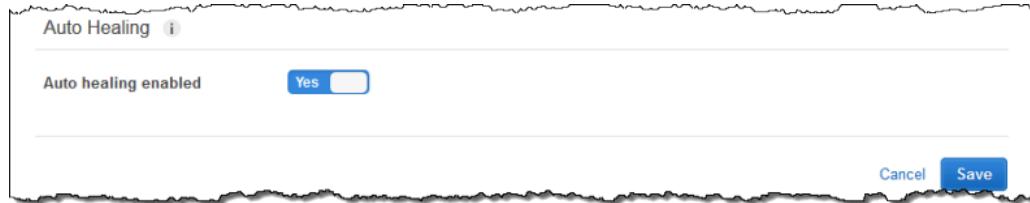
If you [specify an Amazon EBS volume \(p. 63\)](#) for a layer's instances, AWS OpsWorks creates a new volume and attaches it to each instance when the instance is started. If you later want to detach the volume from an instance, use the [Resources](#) page. If AWS OpsWorks auto heals one of the layer's instances:

- If the volume was attached to the instance when it failed, the volume and its data are preserved and AWS OpsWorks reattaches it to the new instance.
- Otherwise, AWS OpsWorks creates a new, empty volume with the configuration specified by the layer and attaches that volume to the instance.

Auto healing is enabled by default for all layers, but you can change that setting.

### To change a layer's auto healing setting

1. On the **Layers** page, click layer name, click the **General Settings** tab, and then click **Edit** to edit the auto healing settings.



2. For **Auto healing enabled**, click the toggle button to turn auto healing on or off.

#### Important

If you have auto healing enabled, be sure to do the following:

- Use only the AWS OpsWorks console, CLI, or API to stop instances.  
If you stop an instance in any other way, such as using the Amazon EC2 console, AWS OpsWorks will treat the instance as failed and auto heal it.
- Use Amazon EBS volumes to store any data that you don't want to lose if the instance is auto healed.

Auto healing terminates the Amazon EC2 instance, which destroys any data that is not stored on an Amazon EBS volume. Amazon EBS volumes are reattached to the new instance, which preserves any stored data.

## How to Delete an OpsWorks Layer

If you no longer need an AWS OpsWorks layer, you can delete it from your stack.

### To delete an OpsWorks layer

1. In the navigation pane, click **Instances**.
2. On the **Instances** page, under the name of the layer you want to delete, click **stop** in the **Actions** column for each instance.. Click **Yes, stop** when prompted.

## Instances

[Stop All Instances](#)

An instance represents an EC2 instance. Each instance belongs to a layer that defines the instance's settings, resources, installed packages, profiles and security groups. When you start the instance, OpsWorks uses the associated layer's blueprint to create and configure a corresponding EC2 instance. [Learn more](#).



### PHP App Server

Host Name	Status	Size	Type	AZ	Public IP	Actions
php-app1	online	c1.medium	24/7	us-east-1a	54.242.127.207	<span style="color: green;">stop</span>

**Are you sure you want to stop php-app1?**  
All data not stored on EBS volumes will be lost.

Cancel Stop

[+ Instance](#)

3. After each instance has stopped, click **delete** for each to remove it from the layer. Click **Yes, delete** when prompted.

Host Name	Status	Size	Type	AZ	Actions
db-master1	stopped	c1.medium	24/7	us-east-1a	<span style="color: green;">start</span> <span style="color: red;">delete</span>

**Are you sure that you want to delete db-master1?**  
If you delete this instance all logs and data associated with it will be lost.

Cancel Delete

[+ Instance](#)

4. In the navigation pane, click **Layers**.
5. On the **Layers** page, click **delete** in the **Actions** column for the layer.

**PHP App Server**

General Settings / Recipes / Network / EBS Volumes / Security /

Delete

[+ Layer](#)

## Load Balancer Layers

A stack typically requires multiple application server instances to effectively handle incoming application requests. You can improve an application's availability, performance, and scalability by adding a load balancer to your stack. A load balancer exposes a single public IP address to represent the application,

receives all incoming requests, distributes them evenly to the stack's application server instances, receives the responses, and returns them to the caller.

AWS OpsWorks includes two built-in ways to balance your application's load.

### Elastic Load Balancing

Elastic Load Balancing is an AWS service that automatically distributes incoming application traffic across multiple Amazon EC2 instances. Such a layer has several advantages:

- Automatically scales request handling capacity in response to incoming application traffic
- Stores SSL certificates by using IAM credentials, allowing you to control who can see your private keys
- Spans multiple Availability Zones for reliability but provides a single DNS name for simplicity, eliminating the need for techniques such as round robin DNS to provide availability at the load balancer tier
- Uses Amazon CloudWatch to report metrics such as request count and request latency
- Supports SSL termination at the Load Balancer by default, including offloading SSL decryption from application instances, centralized SSL certificate management, and encryption to back-end instances with optional public key authentication

For more information, see [Elastic Load Balancing Developer Guide](#).

### HAProxy

The HAProxy layer uses Chef recipes to install HAProxy on EC2 instances that you run and control. It gives you deep control over the load balancer's functionality, including the ability to customize the configuration by overriding attributes, modifying configuration files, and so on. However, the greater amount of control is accompanied by higher degree of responsibility for issues such as capacity management and configuration for availability. For more information, see [HAProxy](#).

It can sometimes be useful to use both load balancers. For example, you could use an Elastic Load Balancing load balancer to distribute traffic to a set of HAProxy instances in different Availability Zones, each of which distributes its traffic to a set of application servers.

### Topics

- [Elastic Load Balancing \(p. 70\)](#)
- [HAProxy AWS OpsWorks Layer \(p. 72\)](#)

## Elastic Load Balancing

Elastic Load Balancing works somewhat differently than an AWS OpsWorks layer. Instead of creating a layer and adding instances to it, you attach an Elastic Load Balancing load balancer to one of your existing layers. In addition to distributing traffic, Elastic Load Balancing does the following:

- Detects unhealthy Amazon EC2 instances and reroutes traffic to the remaining healthy instances until the unhealthy instances have been restored
- Automatically scales request handling capacity in response to incoming traffic

### Note

When you add an Elastic Load Balancing load balancer to a layer, AWS OpsWorks automatically registers the layer's existing instances with the service, which distributes incoming application traffic across those instances. When you start additional instances for the layer, including [load-based and time-based instances \(p. 113\)](#), AWS OpsWorks automatically registers them with the Elastic Load Balancing service.

To use Elastic Load Balancing with a stack, you must first create one or more load balancers in the same region by using the Elastic Load Balancing console, CLI, or API. You can attach only one Elastic Load Balancing load balancer to a layer, and each load balancer can handle only one layer. This means that you must create a separate Elastic Load Balancing load balancer for each layer in each stack that you want to balance and use it only for that purpose. A recommended practice is to assign a distinctive name to each Elastic Load Balancing load balancer that you plan to use with AWS OpsWorks, such as MyStack1-RailsLayer-ELB, to avoid using a load balancer for more than one purpose.

**Important**

We recommend creating new Elastic Load Balancing load balancers for your AWS OpsWorks layers. If you choose to use an existing Elastic Load Balancing load balancer, you should first confirm that it is not being used for other purposes and has no attached instances. After the load balancer is attached to the layer, OpsWorks removes any existing instances and configures the load balancer to handle only the layer's instances. Although it is technically possible to use the Elastic Load Balancing console or API to modify a load balancer's configuration after attaching it to a layer, you should not do so; the changes will not be permanent.

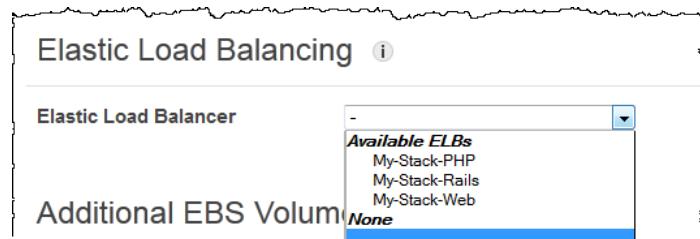
**To attach an Elastic Load Balancing load balancer to a layer**

1. If you have not yet done so, use the [Elastic Load Balancing console](#), API, or CLI to create a load balancer. For more information, see [Elastic Load Balancing](#).

**Tip**

When you create the load balancer, be sure to specify a health check ping path that is appropriate for your application. The default ping path is /index.html. If your application does not use index.html, you must specify an appropriate path or the health check will fail. For example, the SimplePHP application used in [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#) does not use index.html, and the ping path for that app is /. For more information, see [Elastic Load Balancing](#).

2. Create the layer that you want to have balanced. For more information, see [How to Create an OpsWorks Layer \(p. 58\)](#).
3. In the navigation pane, click **Layers**.
4. For the appropriate layer, click **Network** and then click **Edit**.
5. Under **Elastic Load Balancing**, select the load balancer that you want to attach to the layer and click **Save**.



After you have attached the load balancer to the layer, AWS OpsWorks automatically registers the layer's instance's when they come online and deregisters instances when they leave the online state, including load-based and time-based instances. OpsWorks also automatically activates and deactivates the instances' Availability Zones.

**Note**

After an instance has booted, AWS OpsWorks runs the [Setup and Deploy recipes \(p. 175\)](#), which install packages and deploy applications. After those recipes have finished, the instance is in the online state and AWS OpsWorks registers the instance with Elastic Load Balancing. AWS OpsWorks also triggers a Configure event after the instance comes online. This means that Elastic Load Balancing registration and the Configure recipes could run concurrently, and the instance might be registered before the Configure recipes have finished. To ensure that a recipe

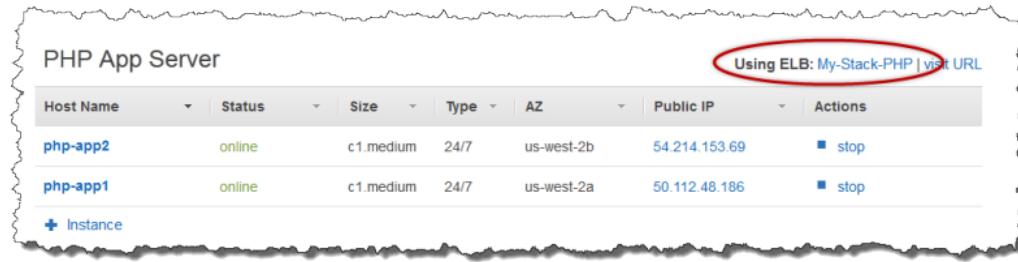
finishes before an instance is registered with Elastic Load Balancing, you should add the recipe to the layer's Setup or Deploy lifecycle events. For more information, see [Executing Recipes \(p. 175\)](#).

You can attach multiple load balancers to a particular set of instances as follows:

### To attach multiple load balancers

1. Use the [Elastic Load Balancing console](#), API, or CLI to create a set of load balancers.
2. [Create a custom layer \(p. 96\)](#) for each load balancer and attach one of the load balancers to it. You don't need to implement any custom recipes for these layers; a default custom layer is sufficient.
3. [Add the set of instances \(p. 103\)](#) to each custom layer.

You can examine a load balancer's properties by going to the Instances page and clicking the appropriate load balancer name.



The **ELB** page shows the load balancer's basic properties, including its DNS name and the health status of the associated instances. If the stack is running in a VPC, the page shows subnets rather than Availability Zones. A green check indicates a healthy instance. You can click on the name to connect to a server, through the load balancer.

## ELB My-Stack-PHP

[Disconnect ELB](#)

Elastic Load Balancing associates your load balancer with your EC2 instances using IP addresses. [Learn more](#).

### Settings

DNS Name                    [My-Stack-PHP-1556928710.us-west-2.elb.amazonaws.com](#)

Layer                        [PHP App Server](#)

Region                      us-west-2

us-west-2a

1

us-west-2b

1

[php-app1](#) ●

✓

[php-app2](#) ●

✓

## HAProxy AWS OpsWorks Layer

The AWS OpsWorks HAProxy layer is an AWS OpsWorks layer that provides a blueprint for instances that host an [HAProxy](#) server—a reliable high-performance TCP/HTTP load balancer. One small instance is usually sufficient to handle all application server traffic.

**Note**

Stacks are limited to a single region. To distribute your application across multiple regions, you must create a separate stack for each region.

**To create a HAProxy layer**

1. In the navigation pane, click **Layers**.
2. On the **Layers** page, click **Add a Layer** or **+ Layer**. For **Layer type**, select **HAProxy**.

The layer has the following configuration settings, all of which are optional.

**HAProxy statistics**

Whether the layer collects and displays statistics. The default value is **On**.

**Statistics URL**

The statistics page's URL path. The complete URL is `http://DNSNameStatisticsPath`, where *DNSName* is the associated instance's DNS name. The default *StatisticsPath* value is `/haproxy?stats`, which corresponds to something like: `http://ec2-54-245-151-7.us-west-2.compute.amazonaws.com/haproxy?stats`.

**Statistics user name**

The statistics page's user name, which you must provide to view the statistics page. The default value is "opsworks".

**Statistics password**

A statistics page password, which you must provide to view the statistics page. The default value is a randomly generated string.

**Health check URL**

The health check URL suffix. HAProxy uses this URL to periodically call an HTTP method on each application server instance to determine whether the instance is functioning. If the health check fails, HAProxy stops routing traffic to the instance until it is restarted, either manually or through [auto healing \(p. 67\)](#). The default value for the URL suffix is "/", which corresponds to the server instance's home page: `http://DNSName/`.

**Health check method**

An HTTP method to be used to check whether instances are functioning. The default value is **OPTIONS** and you can also specify **GET** or **HEAD**. For more information, see [httpchk](#).

**Custom security groups**

This setting appears if you chose to not automatically associate a built-in AWS OpsWorks security group with your layers. You must specify which security group to associate with the layer. Make sure that the group has the correct settings to allow traffic between layers. For more information, see [Create a New Stack \(p. 41\)](#).

## Add Layer

OpsWorks RDS

Layer type: **HAProxy** Looking for a different Layer type? [Let us know.](#)

An HAProxy layer is a blueprint for instances that expose a single IP address to represent a set of application servers. It receives incoming requests, distributes them across the application server instances, and returns responses to the caller. [Learn more.](#)

HAProxy statistics	<input checked="" type="checkbox"/> Yes
Statistics URL	/haproxy?stats
Statistics user name	opsworks
Statistics password	nbok8ypxjk
Health check URL	/
Health check method	OPTIONS

**Add Layer**

### Note

Record the password for later use; AWS OpsWorks does not allow you to view the password after you create the layer. However, you can reset the password by going to the layer's **Edit** page and clicking **Reset now** on the **General Settings** tab.

## Layer HAProxy

General Settings Recipes Network EBS Volumes Security

**Settings**

HAProxy statistics	<input checked="" type="checkbox"/> Yes
Statistics URL	/haproxy?stats
Statistics user name	opsworks
Statistics password	Your password has already been set <a href="#">Reset now.</a>
Health check URL	/
Health check method	OPTIONS

**Auto Healing** i

Auto healing enabled

## How the HAProxy Layer Works

By default, HAProxy does the following:

- Listens for requests on the HTTP and HTTPS ports.

You can configure HAProxy to listen on only the HTTP or HTTPS port by overriding the Chef configuration template, `haproxy.cfg.erb`.

- Routes incoming traffic to instances that are members of any application server layer.

By default, AWS OpsWorks configures HAProxy to distribute traffic to instances that are members of any application server layer. You could, for example, have a stack with both Rails App Server and PHP App Server layers, and an HAProxy master distributes traffic to the instances in both layers. You can configure the default routing by using a custom recipe.

- Routes traffic across multiple Availability Zones.

If one Availability Zone goes down, the load balancer routes incoming traffic to instances in other zones so your application continues to run without interruption. For this reason, a recommended practice is to distribute your application servers across multiple Availability Zones.

- Periodically runs the specified health check method on each application server instance to assess its health.

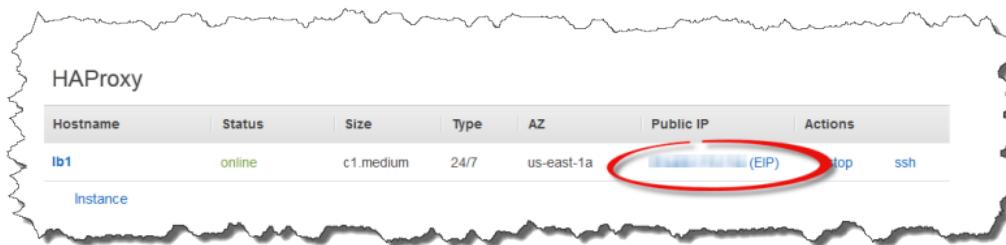
If the method does not return within a specified timeout period, the instance is presumed to have failed and HAProxy stops routing requests to the instance. AWS OpsWorks also provides a way to automatically replace failed instances. For more information, see [How to Use Auto Healing to Replace a Layer's Failed Instances \(p. 67\)](#). You can change the health check method when you create the layer.

- Collects statistics and optionally displays them on a web page.

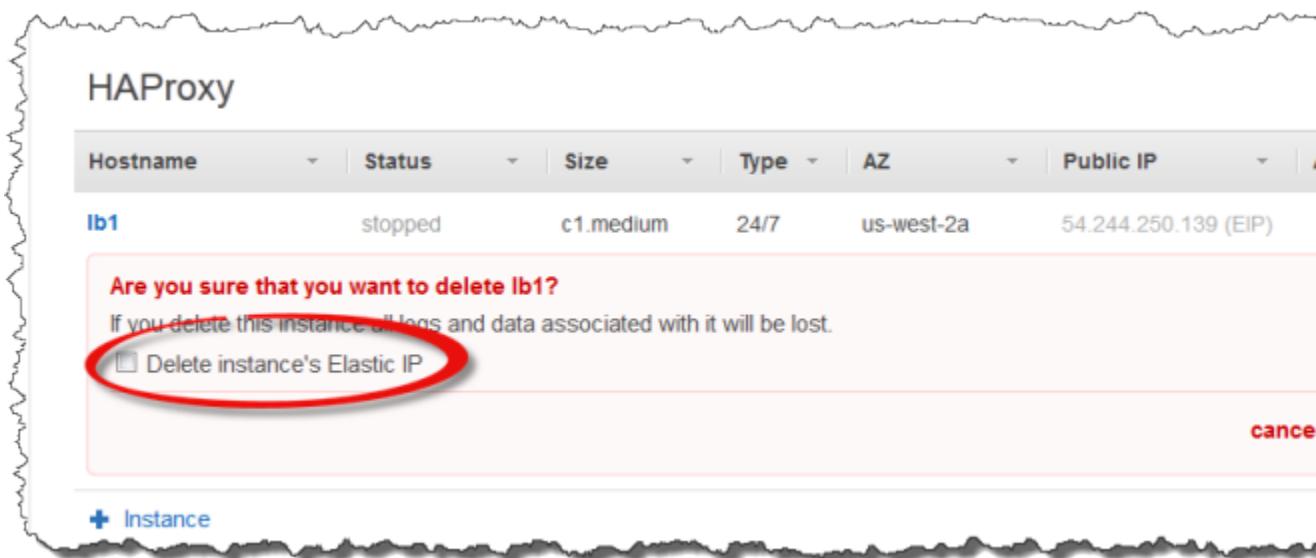
### Important

For health check to work correctly with the default OPTIONS method, your app must return a 2xx or 3xx status code.

By default, when you add an instance to a HAProxy layer, AWS OpsWorks assigns it an Elastic IP address to represent the application, which is public to the world. Because the HAProxy instance's Elastic IP address is the application's only publicly exposed URL, you don't have to create and manage public domain names for the underlying application server instances. You can obtain the address by going to the **Instances** page and examining the instance's public IP address, as the following illustration shows. An address that is followed by (EIP) is an Elastic IP address. For more information on Elastic IP addresses, see [Elastic IP Addresses \(EIP\)](#).



When you stop an HAProxy instance, AWS OpsWorks retains the Elastic IP address and reassigns it to the instance when you restart it. If you delete an HAProxy instance, by default, AWS OpsWorks deletes the instance's IP address. To retain the address, clear the Delete instance's Elastic IP option, as shown in the following illustration.



This option affects what happens when you add a new instance to the layer to replace a deleted instance:

- If you retained the deleted instance's Elastic IP address, AWS OpsWorks assigns the address to the new instance.
- Otherwise, AWS OpsWorks assigns a new Elastic IP address to the instance and you must update your DNS registrar settings to map to the new address.

When application server instances come on line or go off line—either manually or as a consequence of [auto scaling](#) (p. 113) or [auto healing](#) (p. 67)—the load balancer configuration must be updated to route traffic to the current set of online instances. This task is handled automatically by the layer's built-in recipes:

- When new instances come on line, AWS OpsWorks triggers a [Configure lifecycle event](#) (p. 176). The HAProxy layer's built-in Configure recipes update the load balancer configuration so that it also distributes requests to any new application server instances.
- When instances go off line or an instance fails a health check, AWS OpsWorks also triggers a [Configure lifecycle event](#). The HAProxy Configure recipes update the load balancer configuration to route traffic to only the remaining online instances.

Finally, you can also use a custom domain with the HAProxy layer. For more information, see [Using Custom Domains](#) (p. 146).

## Statistics Page

If you have enabled the statistics page, the HAProxy displays a page containing a variety of metrics at the specified URL.

### To view HAProxy statistics

1. On the **Layers** page, click **HAProxy** to open the layer's details page.
2. Obtain the HAProxy instance's **Public DNS** name from the instance's **Details** page and append the statistics URL to it. For example: `http://ec2-54-245-102-172.us-west-2.compute.amazonaws.com/haproxy?stats`.
3. Paste the URL from the previous step into your browser and use the user name and password that you specified when you created the layer to open the statistics page.

## HAProxy version 1.4.22, released 2012/08/09

### Statistics Report for pid 2468

#### > General process information

```
pid = 2468 (process #1, nbproc = 1)
uptime = 0d 2h48m51s
system limits: memmax = unlimited; ulimit-n = 160013
maxsock = 160013; maxconn = 80000; maxpipes = 0
current connns = 1; current pipes = 0/0
Running tasks: 1/2
```

active UP      backup UP  
 active UP, going down      backup UP, going down  
 active DOWN, going up      backup DOWN, going up  
 active or backup DOWN      not checked  
 active or backup DOWN for maintenance (M)

Note: UP with load-balancing disabled is reported as

application	Queue			Session rate			Sessions				Bytes		Denied		Errors			
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out	Req	Resp	Req	Conn	Resp
Frontend				1	1	-	1	1	80 000	2		335	262	0	0	0	0	
localhost	0	0	-	0	0		0	0	5	0	0	0	0	0	0	0	0	
Backend	0	0		0	0		0	0	80 000	0	0	335	262	0	0	0	0	

# Database Layers

AWS OpsWorks provides built-in layers for two types of database servers:

- An Amazon RDS service layer represents an [Amazon RDS database instance](#).

An Amazon RDS instance can host a MySQL, Oracle, Microsoft SQL Server, or PostgreSQL database server and also manages administrative tasks. A stack can include any number of Amazon RDS service layers.

- A MySQL OpsWorks layer is a blueprint for creating instances that host a MySQL database master.

A stack can have only one MySQL layer, but you can add any number of instances to the layer, each of which hosts a separate MySQL master on an Amazon EC2 instance.

After you add a database server to your stack, you can [associate it with an app](#) (p. 125), which enables the application to [set up a connection with the database server](#) (p. 135).

#### Tip

You can incorporate other types of database servers into your stack by using custom JSON or by implementing a custom layer. For more information, see [Using Other Back-end Data Stores](#) (p. 300).

#### Topics

- [Amazon RDS Service Layer](#) (p. 77)
- [MySQL OpsWorks Layer](#) (p. 80)

## Amazon RDS Service Layer

An Amazon RDS service layer represents an Amazon RDS instance. The layer can represent only existing Amazon RDS instances, which you must create separately by using the [Amazon RDS console](#) or API.

The basic procedure for incorporating an Amazon RDS service layer into your stack is as follows:

1. Use the Amazon RDS console, API, or CLI to create an instance.

Be sure to record the instance's ID, master user name, master password, and database name.

2. To add an Amazon RDS service layer to your stack, register the Amazon RDS instance with the stack.
3. Attach the service layer to an app, which adds the Amazon RDS instance's connection information to the app's [deployment JSON \(p. 265\)](#).
4. Use the language-specific files or the information in the deployment JSON to connect the application to the Amazon RDS.

For more information on how to connect an application to a database server, see the section called “[Connecting an Application to a Database Server” \(p. 135\)](#)

#### Topics

- [Specifying Security Groups \(p. 78\)](#)
- [Registering an Amazon RDS Instance with a Stack \(p. 78\)](#)
- [Associating Amazon RDS Service Layers with Apps \(p. 80\)](#)
- [Removing an Amazon RDS Service Layer from a Stack \(p. 80\)](#)

## Specifying Security Groups

To use an Amazon RDS instance with AWS OpsWorks, its database or VPC security groups must allow access from the appropriate IP addresses. For production use, a security group usually limits access to only those IP addresses that need to access the database. It typically includes the addresses of the systems that you use to manage the database and the AWS OpsWorks instances that need to access the database. AWS OpsWorks automatically creates an Amazon EC2 security group for each type of layer when you create your first stack. A simple way to provide access for AWS OpsWorks instances is to add the appropriate layer's security group to the database or VPC security groups, as follows:

#### To add a layer's security group to a database security group

1. Open the Amazon RDS console at <https://console.aws.amazon.com/rds/>.
2. Click **Security Groups** in the navigation pane, select the appropriate group, and then click **Edit**.
3. Under **Connection Type**, select **EC2 Security Groups**.
4. Select the layer's security group from the **EC2 Security Group** list in the **Details** column and click **Add**. For example, to provide database access to PHP App Server instances, select **AWS-OpsWorks-PHP-App-Server (*security\_group\_id*)**.

## Registering an Amazon RDS Instance with a Stack

To add an Amazon RDS service layer in a stack, you must register an instance with the stack.

#### To register an Amazon RDS instance with a stack

1. In the AWS OpsWorks console, click **Layer** in the navigation pane, click **+ Layer** or **Add a layer** to open the **Add Layer** page, and then click the **RDS** tab.
2. If necessary, update the stack's service role, as described in [Updating the Stack's Service Role \(p. 79\)](#).
3. Click the **RDS** tab to list the available Amazon RDS instances.

#### Tip

If your account does not have any Amazon RDS instances, you can create one by clicking **Add an RDS instance** on the RDS tab, which takes you to the Amazon RDS console and starts the **Launch a DB Instance** wizard. You can also go directly to the [Amazon RDS console](#) and click **Launch a DB Instance**, or use the Amazon RDS API or CLI. For more information on how to create an Amazon RDS instance, see [Getting Started with Amazon RDS](#).

- Select the appropriate instance, set **User** and **Password** to the appropriate user and password values and click **Register to Stack**.

**Important**

You must ensure that the user and password that you use to register the Amazon RDS instance correspond to a valid user and password. If they do not, your applications will not be able connect to the instance. However, you can [edit the layer \(p. 59\)](#) to provide valid user and password values and then redeploy the app.

## Add Layer

Instance Identifier	Engine	Storage (GB)	Type	Status	Multi-AZ	Availability Zone
opsinstance2	mysql	5	t1.micro	available	No	us-east-1a

Connection Details for opsinstance2

User: opsuser

Password: [REDACTED] [SHOW](#)

Please verify that OpsWorks can connect to your RDS Instance by setting [Security Groups](#) on that instance. [Learn more](#).

[Cancel](#) [Register with Stack](#)

When you add an Amazon RDS service layer to a stack, AWS OpsWorks assigns it an ID and adds the associated Amazon RDS configuration to the [stack configuration and deployment JSON's \(p. 262\)](#) `[ :opsworks ][:stack] (p. 386)` attribute.

**Note**

If you change a registered Amazon RDS instance's password, you must manually update the password in AWS OpsWorks and then redeploy your apps to update the stack configuration and deployment JSON on the stack's instances.

**Topics**

- [Updating the Stack's Service Role \(p. 79\)](#)

## Updating the Stack's Service Role

Every stack has an [IAM service role \(p. 293\)](#) that specifies what actions AWS OpsWorks can perform on your behalf with other AWS services. To register an Amazon RDS instance with a stack, its service role must grant AWS OpsWorks permissions to access Amazon RDS.

The first time you add an Amazon RDS service layer to one of your stacks, the service role might lack the required permissions. If so, when you click the RDS tab on the **Add Layer** page, you will see the following.

## Add Layer

To use RDS instances, your OpsWorks IAM role needs to have an RDS instances access policy.

[Update](#)

Click **Update** to have AWS OpsWorks update the service role's policy to the following.

```
{"Statement": [ {"Action": ["ec2:*", "iam:PassRole", "cloudwatch:GetMetricStatistics", "elasticloadbalancing:*", "rds:*"], "Effect": "Allow", "Resource": ["*"] } ]}
```

**Note**

You need to perform the update only once. The updated role is then automatically used by all of your stacks.

## Associating Amazon RDS Service Layers with Apps

After you add an Amazon RDS service layer, you can associate it with an app.

- You can associate an Amazon RDS layer to an app when you [create the app \(p. 125\)](#), or later by [editing the app's configuration \(p. 134\)](#).
- To disassociate an Amazon RDS layer from an app, edit the app's configuration to specify a different database server, or no server.

The Amazon RDS layer remains part of the stack, and can be associated with a different app.

After you associate an Amazon RDS instance with an app, AWS OpsWorks puts the database connection information on the app's servers. The application on each server instance can then use this information to connect to the database. For more information on how to connect to an Amazon RDS instance, see the section called [“Connecting an Application to a Database Server” \(p. 135\)](#).

## Removing an Amazon RDS Service Layer from a Stack

To remove an Amazon RDS service layer from a stack, you deregister it.

### To deregister an Amazon RDS service layer

1. Click **Layers** in the navigation pane and click the Amazon RDS service layer's name.
2. Click **Deregister** and confirm that you want to deregister the layer.

This procedure removes the layer from the stack, but it does not delete the underlying Amazon RDS instance. The instance and any databases remain in your account and can be registered with other stacks. You must use the Amazon RDS console, API, or CLI to delete the instance. For more information, see [Deleting a DB Instance](#).

## MySQL OpsWorks Layer

A MySQL OpsWorks layer provides a blueprint for Amazon EC2 instances that function as a MySQL database master. A built-in recipe creates a database for each application that has been deployed to an application server layer. For example, if you deploy a PHP application “myapp,” the recipe creates a “myapp” database.

The MySQL layer has the following configuration settings.

#### MySQL root user password

(Required) The root user password.

#### Set root user password on every instance

(Optional) Whether the root user password is included in the stack configuration JSON that is passed to every instance in the stack. The default setting is **Yes**.

If you set this value to **No**, AWS OpsWorks passes the root password only to application server instances.

#### Custom security groups

(Optional) A custom security group to be associated with the layer. For more information, see [Create a New Stack \(p. 41\)](#).

### Add layer

The screenshot shows the 'Add layer' dialog. It has a dropdown menu for 'Layer type' set to 'MySQL'. Below it is a descriptive text: 'A MySQL Master layer is a blueprint for instances that function as MySQL relational database servers.' followed by a 'Learn more' link. The 'MySQL root user password' field contains 'y3g19ptugl'. The 'Set root user password on every instance' checkbox is checked. At the bottom right are 'Cancel' and 'Add layer' buttons.

You can add one or more instances to the layer, each of which represents a separate MySQL database master. You can then [attach an instance to an app \(p. 125\)](#), which installs the necessary connection information on the app's application servers. The application can then use the connection information to [connect to the instance's database server \(p. 135\)](#).

## Application Server Layers

AWS OpsWorks supports several different application servers, where "application" includes static web pages. Each type of server has a separate AWS OpsWorks layer, with built-in recipes that handle installing the application server and any related packages on each of the layer's instances, deploying apps, and so on. For example, the Java App Server layer installs several packages—including Apache, Tomcat, and OpenJDK—and deploys Java apps to each of the layer's instances.

The following is the basic procedure for using an application server layers:

1. [Create \(p. 58\)](#) one of the five available **App Server** layer types.
2. [Add one or more instances \(p. 103\)](#) to the layer.
3. Create apps and deploy them to the instances. For more information, see [Apps \(p. 125\)](#).
4. (Optional) If the layer has multiple instances, you can add a load balancer, which distributes incoming traffic across the instances. For more information, see [HAProxy AWS OpsWorks Layer \(p. 72\)](#).

#### Topics

- [Java App Server AWS OpsWorks Layer \(p. 82\)](#)
- [Node.js App Server AWS OpsWorks Layer \(p. 88\)](#)
- [PHP App Server AWS OpsWorks Layer \(p. 91\)](#)
- [Rails App Server AWS OpsWorks Layer \(p. 92\)](#)
- [Static Web Server AWS OpsWorks Layer \(p. 95\)](#)

## Java App Server AWS OpsWorks Layer

The Java App Server layer is an AWS OpsWorks layer that provides a blueprint for instances that function as Java application servers. This layer is based on [Apache Tomcat 7.0](#) and [Open JDK 7](#). AWS OpsWorks also installs the Java connector library, which allows Java apps to use a JDBC `DataSource` object to connect to a back end data store.

**Installation:** Tomcat is installed in `/usr/share/tomcat7`.

The **Add Layer** page provides the following configuration options:

### Java VM Options

You can use this setting to specify custom Java VM options; there are no default options. For example, a common set of options is `-Djava.awt.headless=true -Xmx128m -XX:+UseConcMarkSweepGC`. If you use **Java VM Options**, make sure that you pass a valid set of options; AWS OpsWorks does not validate the string. If you attempt to pass an invalid option, the Tomcat server typically fails to start, which causes setup to fail. If that happens, you can examine the instance's setup Chef log for details. For more information on how to view and interpret Chef logs, see [Chef Logs \(p. 338\)](#).

### Custom security groups

This setting appears if you chose to not automatically associate a built-in AWS OpsWorks security group with your layers. You must specify which security group to associate with the layer. For more information, see [Create a New Stack \(p. 41\)](#).

### Elastic Load Balancer

You can attach an Elastic Load Balancing load balancer to the layer's instances. For more information, see [the section called “Elastic Load Balancing” \(p. 70\)](#).

## Add Layer

The screenshot shows the 'Add Layer' configuration page. At the top, there are two tabs: 'OpsWorks' (selected) and 'RDS'. Below the tabs, the 'Layer type' dropdown is set to 'Java App Server', with a note 'Looking for a different Layer type? [Let us know.](#)'. A descriptive text states 'Java layer is a blueprint for instances that function as Tomcat web servers.' with a 'Learn more' link. The 'Java version' is set to 'OpenJDK 7'. The 'Java stack' is set to 'Tomcat 7'. The 'Java VM Options' field is empty. The 'Elastic Load Balancer' section indicates 'No ELBs have been created in us-east-1. To add an ELB go to the [EC2 console](#)'. At the bottom right are 'Cancel' and 'Add Layer' buttons.

You can specify other configuration settings by using custom JSON or a custom attributes file. For more information, see [Custom Configuration \(p. 83\)](#).

### Important

The Java App Server layer can be used only with a Chef 11.4 stack. For more information on specifying a stack's Chef version, see [Create a New Stack \(p. 41\)](#) and [Update a Stack \(p. 50\)](#). For guidance on migrating to Chef 11.4, see [Migrating an Existing Stack to a new Chef Version \(p. 169\)](#).

### Topics

- [Custom Configuration \(p. 83\)](#)
- [Deploying Java Apps \(p. 83\)](#)

## Custom Configuration

AWS OpsWorks exposes additional configuration settings as built-in attributes, which are all in the `opsworks_java` namespace. You can use custom JSON or a custom attributes file to override the built-in attributes and specify custom values. For example, the JVM and Tomcat versions are represented by the built-in `jvm_version` and `java_app_server_version` attributes, both of which are set to 7. You can use custom JSON or a custom attributes file to set either or both to 6. The following example uses custom JSON to set both attributes to 6:

```
{  
  "opsworks_java": {  
    "jvm_version": 6,  
    "java_app_server_version": 6  
  }  
}
```

For more information, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#).

Another example of custom configuration is installing a custom JDK by overriding the `use_custom_pkg_location`, `custom_pkg_location_url_debian`, and `custom_pkg_location_url_rhel` attributes.

**Note**

If you override the built-in cookbooks, you will need to update those components yourself.

For more information on attributes and how to override them, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#). For a list of built in attributes, see [opsworks\\_java Attributes \(p. 418\)](#).

## Deploying Java Apps

The following topics describe how to deploy apps to a Java App Server layer's instances. The examples are for JSP apps, but you can use essentially the same procedure for installing other types of Java app.

You can deploy JSP pages from any of the supported repositories. If you want to deploy WAR files, note that AWS OpsWorks automatically extracts WAR files that are deployed from an Amazon S3 or HTTP archive, but not from a Git or Subversion repository. If you want to use Git or Subversion for WAR files, you can do one of the following:

- Store the extracted archive in the repository.
- Store the WAR file in the repository and use a Chef deployment hook to extract the archive, as described in the following example.

You can use Chef deployment hooks to run user-supplied Ruby applications on an instance at any of four deployment stages. The application name determines the stage. The following is an example of a Ruby application named `before_migrate.rb`, which extracts a WAR file that has been deployed from a Git or Subversion repository. The name associates the application with the Checkout deployment hook so it runs at the beginning of the deployment operation, after the code is checked but before migration. For more information on how to use this example, see [Using Chef Deployment Hooks \(p. 238\)](#).

```
::Dir.glob(::File.join(release_path, '*.war')) do |archive_file|  
  execute "unzip_#{archive_file}" do  
    command "unzip #{archive_file}"  
    cwd release_path
```

```
    end
```

### Topics

- [Deploying a JSP App \(p. 84\)](#)
- [Deploying a JSP App with a Database Connection \(p. 85\)](#)

## Deploying a JSP App

To deploy a JSP app, specify the name and repository information. You can also optionally specify domains and SSL settings. For more information on how to create an app, see [Adding Apps \(p. 125\)](#). The following procedure shows how to create and deploy a simple JSP page from a public Amazon S3 archive. For information on how to use other repository types, including private Amazon S3 archives, see [Application Source \(p. 129\)](#).

The following example shows the JSP page, which simply displays some system information.

```
<%@ page import="java.net.InetAddress" %>
<html>
<body>
<%
    java.util.Date date = new java.util.Date();
    InetAddress inetAddress = InetAddress.getLocalHost();
%>
The time is
<%
    out.println( date );
    out.println("<br>Your server's hostname is "+inetAddress.getHostName());
%>
<br>
</body>
</html>
```

### Note

The following procedure assumes that you are already familiar with the basics of creating stacks, adding instances to layers, and so on. If you are new to AWS OpsWorks, you should first see [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#).

### To deploy a JSP page from an Amazon S3 archive

1. [Create a stack \(p. 41\)](#) with a Java App Server layer, [add a 24/7 instance \(p. 103\)](#) to the layer, and [start it \(p. 109\)](#).
2. Copy the code to a file named `simplejsp.jsp`, put the file in a folder named `simplejsp`, and create a `.zip` archive of the folder. The names are arbitrary; you can use any file or folder names that you want. You can also use other types of archive, including `gzip`, `bzip2`, `tarball`, or `Java WAR` file. To deploy multiple JSP pages, include them in the same archive.
3. Upload the archive to an Amazon S3 bucket and make the file public. Copy the file's URL for later use. For more information on how to create buckets and upload files, go to [Get Started With Amazon Simple Storage Service](#).
4. [Add an app \(p. 126\)](#) to the stack and specify following settings:
  - **Name** – SimpleJSP
  - **App type** – Java

- **Repository type** – Http Archive
- **Repository URL** – the Amazon S3 URL of your archive file. It should look something like `https://s3.amazonaws.com/jsp_example/simplejsp.zip`.

Use the default values for the remaining settings and then click **Add App** to create the app.

5. [Deploy the app \(p. 132\)](#) to the Java App Server instance.

You can now go to the app's URL and view the app. If you have not specified a domain, you can construct a URL by using either the instance's public IP address or its public DNS name. To get an instance's public IP address or public DNS name, go the AWS OpsWorks console and click the instance's name on the **Instances** page to open its details page.

The rest of URL depends on the app's short name, which is a lowercase name that AWS OpsWorks generates from the app name that you specified when you created the app. For example the short name of SimpleJSP is simplejsp. You can get an app's short name from its details page.

- If the short name is `root`, you can use either `http://public_DNS/appname.jsp` or `http://public_IP/appname.jsp`.
- Otherwise, you can use either `http://public_DNS/app_shortname/appname.jsp` or `http://public_IP/app_shortname/appname.jsp`.

If you have specified a domain for the app, the URL is `http://domain/appname.jsp`.

The URL for the example would be something like `http://192.0.2.0/simplejsp/simplejsp.jsp`.

If you want to deploy multiple apps to the same instance, you should not use `root` as a short name. This can cause URL conflicts that prevent the app from working properly. Instead, assign a different domain name to each app.

## Deploying a JSP App with a Database Connection

JSP pages can use a JDBC `DataSource` object to connect to a back end database. You create and deploy such an app by using the procedure in the previous section, with one additional step to set up the connection.

The following JSP page shows how to connect to a `DataSource` object.

```
<html>
  <head>
    <title>DB Access</title>
  </head>
  <body>
    <%@ page language="java" import="java.sql.* , javax.naming.* , javax.sql.*" %>
    <%
      StringBuffer output = new StringBuffer();
      DataSource ds = null;
      Connection con = null;
      Statement stmt = null;
      ResultSet rs = null;
      try {
        Context initCtx = new InitialContext();
        ds = (DataSource) initCtx.lookup("java:comp/env/jdbc/mydb");
        con = ds.getConnection();
      }
    %>
```

```
        output.append("Databases found:<br>");  
        stmt = con.createStatement();  
        rs = stmt.executeQuery("show databases");  
        while (rs.next()) {  
            output.append(rs.getString(1));  
            output.append("<br>");  
        }  
    }  
    catch (Exception e) {  
        output.append("Exception: ");  
        output.append(e.getMessage());  
        output.append("<br>");  
    }  
    finally {  
        try {  
            if (rs != null) {  
                rs.close();  
            }  
            if (stmt != null) {  
                stmt.close();  
            }  
            if (con != null) {  
                con.close();  
            }  
        }  
        catch (Exception e) {  
            output.append("Exception (during close of connection): ");  
            output.append(e.getMessage());  
            output.append("<br>");  
        }  
    }  
%>  
<%= output.toString() %>  
</body>  
</html>
```

AWS OpsWorks creates and initializes the `DataSource` object, binds it to a logical name, and registers the name with a Java Naming and Directory Interface (JNDI) naming service. The complete logical name is `java:comp/env/user-assigned-name`. You must specify the user-assigned part of the name by adding custom JSON to the stack and configuration JSON to define the `[ 'opsworks_java' ][ 'data-sources' ]` attribute, as described in the following.

### To deploy a JSP page that connects to a MySQL database

1. [Create a stack \(p. 41\)](#) with a Java App Server layer, [add 24/7 instances \(p. 103\)](#) to each layer, and [start it \(p. 109\)](#).
2. Add a database layer to the stack. The details depend on which type of database you use.

To use a MySQL instance, [add a MySQL layer \(p. 80\)](#) to the stack, add a 24/7 instance to the layer, and start it.

To use an Amazon RDS (MySQL) instance, make sure it has the following configuration:

- The database engine must be MySQL.

- The database security group must include the Java App Server layer's Amazon EC2 security group. AWS OpsWorks creates this security group for you, and it is typically named something like `aws-opsworks-java-app-server`.
- The instance must include a database named `simplejspdb`.

You must then register [register the instance with your stack \(p. 78\)](#).

3. Copy the example code to a file named `simplejspdb.jsp`, put the file in a folder named `simplejspdb`, and create a `.zip` archive of the folder. The names are arbitrary; you can use any file or folder names that you want. You can also use other types of archive, including `gzip`, `bzip2`, or `tarball`. To deploy multiple JSP pages, include them in the same archive.
4. Upload the archive to an Amazon S3 bucket and make the file public. Copy the file's URL for later use. For more information on how to create buckets and upload files, go to [Get Started With Amazon Simple Storage Service](#).
5. [Add an app \(p. 126\)](#) to the stack and specify following settings:
  - **Name** – `SimpleJSPDB`
  - **App type** – `Java`
  - **Data source type** – **OpsWorks** (for a MySQL instance) or **RDS** (for an Amazon RDS instance).
  - **Database instance** – The MySQL instance you created earlier, which is typically named `db-master1(mysql)`, or the Amazon RDS instance, which will be named `DB_instance_name(mysql)`.
  - **Database name** – `simplejspdb`.
  - **Repository type** – `Http Archive`
  - **Repository URL** – the Amazon S3 URL of your archive file. It should look something like `https://s3.amazonaws.com/jsp_example/simplejspdb.zip`.

Use the default values for the remaining settings and then click **Add App** to create the app.

6. Add the following custom JSON to the stack configuration JSON, where `simplejspdb` is the app's short name.

```
{
  "opsworks_java": {
    "datasources": {
      "simplejspdb": "jdbc/mydb"
    }
  }
}
```

AWS OpsWorks uses this mapping to generate a context file with the necessary database information.

For more information on how to add custom JSON to the stack configuration JSON, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#).

7. [Deploy the app \(p. 132\)](#) to the Java App Server instance.

You can now use the app's URL to view the app. For a description of how to construct the URL, see [Deploying a JSP App \(p. 84\)](#).

The URL for the example would be something like `http://192.0.2.0/simplejspdb/simplejspdb.jsp`.

#### Note

The `datasources` attribute can contain multiple attributes. Each attribute is named with an app's short name and set to the appropriate user-assigned part of a logical name. If you have multiple

apps, you can use separate logical names, which requires a custom JSON something like the following.

```
{  
    "opsworks_java": {  
        "datasources": {  
            "myjavaapp": "jdbc/myappdb",  
            "simplejsp": "jdbc/myjsedb",  
            ...  
        }  
    }  
}
```

## Node.js App Server AWS OpsWorks Layer

The Node.js App Server layer is an AWS OpsWorks layer that provides a blueprint for instances that function as [Node.js](#) application servers. AWS OpsWorks also installs [Express](#), so the layer's instances support both standard and Express applications.

**Installation:** Node.js is installed in `/usr/local/bin/node`.

The [Add Layer](#) page provides the following configuration options:

### Node.js version

The default value is 0.10.27.

### Custom security groups

This setting appears if you chose to not automatically associate a built-in AWS OpsWorks security group with your layers. You must specify which security group to associate with the layer. For more information, see [Create a New Stack \(p. 41\)](#).

### Elastic Load Balancer

You can attach an Elastic Load Balancing load balancer to the layer's instances.

## Add Layer

OpsWorks RDS

Layer type: Node.js App Server Looking for a different Layer type? [Let us know.](#)

The Node.js Application Server layer is a blueprint for instances that function as JavaScript application servers. [Learn more.](#)

Node.js version: 0.10.25

Elastic Load Balancer: No ELBs have been created in us-east-1. To add an ELB go to the [EC2 console](#).

Cancel [Add Layer](#)

## Deploying Node.js Apps

To implement Node.js applications for AWS OpsWorks make sure you meet the following conditions:

- The main file must be named `server.js` and reside in the deployed application's root directory.
- Express apps must include a `package.json` file in the application's root directory.
- The application must listen on port 80 (for HTTP requests) or port 443 (for HTTPS requests).

If you configure a Node.js app to support SSL, AWS OpsWorks does the following.

- Sets the port to 443.
- Puts the specified certificate in a file named `ssl.crt` in your deploy directory's `/shared/config` directory.
- Puts the specified key in a file named `ssl.key` in your deploy directory's `/shared/config` directory.
- Puts the chain certificate, if you have specified one, in a file named `ssl.ca` in your deploy directory's `/shared/config` directory.

Your app can obtain the SSL key and certificates from those files.

**Note**

Express applications commonly use the following code to set the listening port, where `process.env.PORT` represents the default port and resolves to 80:

```
app.set('port', process.env.PORT || 3000);
```

With AWS OpsWorks, you must explicitly specify port 80, as follows:

```
app.set('port', 80);
```

The following shows how to create a basic stack and deploy a simple Express application named helloworld. You can use the same procedure for standard Node.js apps, except that you don't need a `package.json` file. If you are not already familiar with the basics of how to use AWS OpsWorks, you should first read [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#).

## The helloworld application

The helloworld example application consists of the following files.

`server.js` is the main file, and must be in the application's root directory. It sets up an HTTP server and returns an HTML file named `index.html` to the user.

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
    res.sendfile('../index.html');
});

app.use(express.static('public'));

app.listen(80);
console.log('Listening on port 80');
```

`package.json` is the package descriptor, which is required for Express applications and must be in the application's root directory. The helloworld `package.json` has only a few basic attributes. For a complete list of the available attributes, see [package.json](#)

```
{
  "name": "hello-world",
  "description": "hello world test app",
```

```
"version": "0.0.1",
"private": true,
"dependencies": {
  "express": "3.4.4"
}
}
```

`index.html` is the HTML file that `server.js` returns to the user, which prints a simple text message. For convenience, `index.html` is in the root directory, but that is not required.

```
<!DOCTYPE html>
<html>
<head>
  <title>Opsworks Demo Node.js App</title>
</head>
<body>
  <h1>Congratulations!</h1>
  <h2>
    You just deployed your first Express App <br />
    with AWS OpsWorks.
  </h2>
</body>
</html>
```

To use helloworld with AWS OpsWorks, you must put it in a repository. This example uses a public S3 archive, but the procedure is basically the same for the other standard repositories. For information on how to use the other standard repositories, see [Cookbook Repositories \(p. 154\)](#).

### To create helloworld and store it in a public Amazon S3 archive

1. Copy the example code to the specified files and put them in a directory named `helloworld`.
2. Create a `.zip` archive of the `helloworld` directory.
3. Sign in to the AWS Management Console and open the Amazon S3 console at <https://console.aws.amazon.com/s3/>.
4. Create a public bucket named `nodeexample`, and upload `helloworld.zip` to the bucket.  
For a description of how to create a bucket and upload files, see [Get Started With Amazon Simple Storage Service](#).
5. Right-click `helloworld.zip` and click **Make Public** to make the file publicly accessible.
6. Click **Properties** and record the `helloworld.zip` URL for later use. It will look something like `https://s3.amazonaws.com/nodeexample/helloworld.zip`.

## Create a Stack

You can run helloworld on a simple stack with one layer and one instance.

### To create the stack

1. On the AWS OpsWorks dashboard and click **Add Stack** to create a new stack. Assign a name to the stack—the example uses `NodeStack`—, accept the default values for the other settings, and click **Add Stack**.
2. Click **Add a layer**. For **Layer type**, select **Node.js App Server**, and click **Add Layer**.

3. Click **Instances** in the navigation pane, click **Add an instance**, and then click **Add Instance** to accept the defaults. The instance will be named nodejs-app1.
4. Click **Apps** in the navigation pane and then click **Add an app**. Specify the following options and then click **Add App**:
  - **Name**—The app's name, which is used for display purposes; the example uses HelloWorld.
  - **App type**—Set this option to **Node.js**, which directs AWS OpsWorks to deploy the app's code to the Node.js App Server layer's instances.
  - **Repository type**—Set this option to **Http Archive**, which is the appropriate setting for public S3 archives.

There is also an **S3 Archive** repository type, but it is used only for private Amazon S3 archives.

  - **Repository URL**—Set this option to the `helloworld.zip` URL that you recorded earlier.

Use default values for the other settings.

5. Click **Instances** in the navigation pane, click **start** in the nodejs-app1 instance's **Actions** column, and wait for the instance's status to reach **Online**.
6. To run helloworld, click the address in the nodejs-app1 instance's **Public IP** column. You should see something like the following:

## Congratulations!

You just deployed your first Node.js Express App with AWS OpsWorks.

## PHP App Server AWS OpsWorks Layer

The PHP App Server layer is an AWS OpsWorks layer that provides a blueprint for instances that function as PHP application servers. The PHP App Server layer is based on [Apache2](#) with mod\_php and has no standard configuration options.

**Installation:** AWS OpsWorks uses the instance's package installer to install Apache2 and mod\_php in their default locations. For more information about installation, see [Apache](#).

The **Add Layer** page provides the following configuration options:

### Custom security groups

This setting appears if you chose to not automatically associate a built-in AWS OpsWorks security group with your layers. You must specify which security group to associate with the layer. For more information, see [Create a New Stack \(p. 41\)](#).

### Elastic Load Balancer

You can attach an Elastic Load Balancing load balancer to the layer's instances.

## Add Layer

OpsWorks RDS

Layer type: PHP App Server Looking for a different Layer type? [Let us know.](#)

The PHP Application Server layer is a blueprint for instances that function as PHP application servers. By default PHP 5.3 and Apache 2.2 are installed. [Learn more.](#)

Elastic Load Balancer No ELBs have been created in us-east-1. To add an ELB go to the [EC2 console](#).

Cancel Add Layer

### Note

If you want to use PHP 5.5, you must specify Ubuntu 14.04 as your stack's [default operating system \(p. 41\)](#) before you add the PHP App Server layer to your stack. AWS OpsWorks then installs PHP 5.5 by default on all of the layer's instances. If you specify Amazon Linux or Ubuntu 12.04, AWS OpsWorks installs PHP 5.3.

You can modify some Apache configuration settings by using custom JSON or a custom attributes file. For more information, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#). For a list of Apache attributes that can be overridden, see [apache2 Attributes \(p. 400\)](#).

For an example of how to deploy a PHP App, including how to connect the app to a backend database, see [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#).

## Rails App Server AWS OpsWorks Layer

The Rails App Server layer is an AWS OpsWorks layer that provides a blueprint for instances that function as Ruby on Rails application servers.

**Installation:** AWS OpsWorks uses the instance's package installer to install the server packages in their default locations. For more information about Apache/Passenger installation, see [Phusion Passenger](#). For more information about logging, see [Log Files](#). For more information about Nginx/Unicorn installation, see [Unicorn](#).

The **Add Layer** page provides the following configuration options, all of which are optional.

### Ruby Version

The Ruby version that will be used by your apps. The default value is 2.0.0.

### Note

AWS OpsWorks installs a separate Ruby package to be used by recipes and the instance agent. For more information, see [Ruby Versions \(p. 170\)](#).

### Rails Stack

The default Rails stack is [Apache2](#) with [Phusion Passenger](#). You can also use [Nginx](#) with [Unicorn](#).

### Note

If you use Nginx and Unicorn, you must add Unicorn to your app's Gemfile, as in the following example:

```
source 'https://rubygems.org'  
gem 'rails', '3.2.15'  
...  
# Use unicorn as the app server
```

```
gem 'unicorn'  
...
```

### Passenger Version

If you have specified Apache2/Passenger, you must specify the Passenger version. The default value is 4.0.42.

### Rubygems Version

The default [Rubygems](#) version is 2.1.7.

### Install and Manage Bundler

Lets you choose whether to install and manage [Bundler](#). The default value is **Yes**.

### Bundler version

The default Bundler version is 1.5.1.

### Custom security groups

This setting appears if you chose to not automatically associate a built-in AWS OpsWorks security group with your layers. You must specify which security group to associate with the layer. For more information, see [Create a New Stack \(p. 41\)](#).

### Elastic Load Balancer

You can attach an Elastic Load Balancing load balancer to the layer's instances.

## Add Layer

The screenshot shows the 'Add Layer' configuration dialog. At the top, there are two tabs: 'OpsWorks' (selected) and 'RDS'. Below the tabs, the 'Layer type' is set to 'Rails App Server'. A note says 'Looking for a different Layer type? [Let us know.](#)' Below this, a note states: 'The Rails Application Server layer is a blueprint for instances that function as Ruby on Rails application servers. [Learn More.](#)' The configuration fields include:

Ruby version	2.0.0
Rails stack	<input checked="" type="radio"/> Apache2 and Passenger <input type="radio"/> nginx and Unicorn
Passenger version	4.0.42
RubyGems version	2.1.7
Install and manage Bundler	<b>Yes</b>
Bundler version	1.5.1
Elastic Load Balancer	-

At the bottom right are 'Cancel' and 'Add Layer' buttons.

You can modify some configuration settings by using custom JSON or a custom attributes file. For more information, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#). For a list of Apache, Nginx, Phusion Passenger, and Unicorn attributes that can be overridden, see [Built-in Recipe Attributes \(p. 400\)](#).

### Topics

- [Connecting to a Database \(p. 93\)](#)
- [Deploying Ruby on Rails Apps \(p. 94\)](#)

## Connecting to a Database

The Rails App Server Configure recipe uses the app's `database` attribute values from the [deployment JSON \(p. 265\)](#) to generate `database.yml`. If you use a built-in layer to provide your database support, these attribute values are automatically set. For more information, see [Rails App Server \(p. 138\)](#).

If you want to use a different database, you can do so by using custom JSON to specify the following [ "deploy" ] [ "appshortname" ] [ "database" ] attributes, which overrides any default settings. `appshortname` is the app's short name, which AWS OpsWorks generates from the app name. For more information, see [Adding Apps \(p. 125\)](#).

- `adapter` (String)
- `host` (String)
- `database` (String)
- `username` (String)
- `password` (String)
- `reconnect` (Boolean)
- `port` (Number, optional). The default port is set by the adapter.

The custom JSON for an app whose short name is `myapp` will look like:

```
{  
  "deploy" : {  
    "myapp" : {  
      "database" : {  
        "adapter" : "adapter",  
        "database" : "databasename",  
        "host" : "host",  
        "password" : "password",  
        "port" : portnumber,  
        "reconnect" : true/false,  
        "username" : "username"  
      }  
    }  
  }  
}
```

For information on how to specify custom JSON, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#). To see the template used to create `database.yml` (`database.yml.erb`), go to the [built-in cookbook repository](#).

## Deploying Ruby on Rails Apps

You can deploy Ruby on Rails apps from any of the supported repositories. The following shows how to deploy an example Ruby on Rails app to a server running an Apache/Passenger Rails stack. The example code is stored in a public GitHub repository, but the basic procedure is the same for the other supported repositories. For more information on how to create and deploy apps, see [Apps \(p. 125\)](#). To view the example's code, which includes extensive comments, go to <https://github.com/awslabs/opsworks-demo-rails-photo-share-app>.

### To deploy a Ruby on Rails app from a GitHub repository

1. Create a stack (p. 41) with a Rails App Server layer with Apache/Passenger as the Rails stack, add a 24/7 instance (p. 103) to the layer, and start it (p. 109).
2. After the instance is online, add an app (p. 126) to the stack and specify following settings:
  - **Name** – Any name you prefer; the example uses `PhotoPoll`.

AWS OpsWorks uses this name for display purposes, and generates a short name for internal use and to identify the app in the [stack configuration and deployment JSON \(p. 262\)](#). For example, the PhotoPoll short name is photopoll.

- **App type – Ruby on Rails.**
- **Rails environment** – The available environments are determined by the application.

The example app has three: **development**, **test**, and **production**. For this example, set the environment to **development**. See the example code for descriptions of each environment.

- **Repository type** – Any of the supported repository types. Specify **git** for this example
- **Repository URL** – The repository that the code should be deployed from.

For this example, set the URL to `git://github.com/aws-labs/opsworks-demo-rails-photo-share-app`.

Use the default values for the remaining settings and then click **Add App** to create the app.

3. [Deploy the app \(p. 132\)](#) to the Rails App Server instance.
4. When deployment is finished, go to the **Instances** page and click the Rails App Server instance's public IP address. You should see the following:



## Static Web Server AWS OpsWorks Layer

The Static Web Server layer is an AWS OpsWorks layer that provides a template for instances to serve static HTML pages, which can include client-side scripting. This layer is based on [Nginx](#).

**Installation:** Nginx is installed in `/usr/sbin/nginx`.

The **Add Layer** page provides the following configuration options:

### Custom security groups

This setting appears if you chose to not automatically associate a built-in AWS OpsWorks security group with your layers. You must specify which security group to associate with the layer. For more information, see [Create a New Stack \(p. 41\)](#).

### Elastic Load Balancer

You can attach an Elastic Load Balancing load balancer to the layer's instances.

## Add Layer

OpsWorks RDS

Layer type: Static Web Server Looking for a different Layer type? [Let us know.](#)

The Static Web Server layer is a blueprint for instances that serve static HTML pages with nginx (which can include client-side scripting). [Learn more.](#)

Elastic Load Balancer No ELBs have been created in us-east-1. To add an ELB go to the [EC2 console](#).

Cancel Add Layer

You can modify some Nginx configuration settings by using custom JSON or a custom attributes file. For more information, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#). For a list of Apache attributes that can be overridden, see [nginx Attributes \(p. 415\)](#).

## Custom AWS OpsWorks Layers

If the standard OpsWorks layers such as Rails App Server don't suit your requirements, even with [customization \(p. 230\)](#), you can create a custom OpsWorks layer, which gives you almost complete control over which packages are installed and how they are configured. However, the custom layer's built-in recipes perform only some very basic tasks. You must implement a set of custom Chef recipes to handle all the tasks of package installation, app deployment, and so on.

**Tip**

A stack can have only one each of AWS OpsWorks built-in layers, but it can have any number of custom layers.

The custom layer has the following configuration settings.

**Name**

(Required) The layer's name, which is used for display purposes.

**Short name**

(Required) The layer's short name, which is used internally and by recipes. The short name is also used as the name for the directory where your app files are installed. It can have a maximum of 200 characters, which are limited to the alphanumeric characters, '-', '\_', and '.'.

**Custom security groups**

This setting appears if you chose to not automatically associate a built-in AWS OpsWorks security group with your layers. You must specify which security group to associate with the layer. For more information, see [Create a New Stack \(p. 41\)](#).

## Add Layer

The screenshot shows the 'Add Layer' interface. At the top, there are tabs for 'OpsWorks' (selected) and 'RDS'. Below the tabs, the 'Layer type' is set to 'Custom'. A note on the right says 'Looking for a different Layer type? [Let us know.](#)'. Under 'Name', the value 'MyCustomLayer' is entered. Under 'Short name', the value 'mycustomlayer' is entered. At the bottom right, there are 'Cancel' and 'Add Layer' buttons.

The basic procedure for creating a custom layer has the following steps:

1. Implement a custom cookbook that contains the recipes and associated files required to install and configure packages, handle configuration changes, deploy apps, and so on.  
Depending on your requirements, you might also need recipes to handle undeployment and shutdown tasks. For more information, see [Cookbooks and Recipes \(p. 153\)](#).
2. Create a custom layer.
3. Assign your custom recipes to the appropriate lifecycle events.

You then add instances to the layer, start them, and deploy apps to those instances just as you do for the other layers.

**Important**

AWS OpsWorks automatically deploys apps only to the built-in [app server layers \(p. 81\)](#). To deploy apps to a custom layer's instances, you must implement custom recipes to handle the deploy operation and assign them to the layer's Deploy event.

For a detailed walkthrough of how to implement recipes for a custom layer and create a stack that includes the layer, see [Creating a Custom Tomcat Server Layer \(p. 240\)](#). The walkthrough is based on a cookbook that supports a basic Tomcat server layer, but much of the discussion and code, including the deployment code, can be readily adapted to other packages.

Combining custom and built-in layers can sometimes be useful. For example, suppose you want to use a PHP application server as an administrator. You can implement a custom Chef recipe that configures a server as an administrator. However, if you assign the recipe to the PHP App Server layer's Configure event, you will configure every instance as an administrator even though you usually want only one. To configure just one server as an administrator, you could do the following:

- Create a custom layer.
- Implement a recipe to configure a server as an administrator and assign it to the layer's Setup event.
- Make one instance a member of both the PHP App Server and Custom layers.

When you start that instance, the PHP App Server Setup recipes configure the instance as a PHP application server and the custom layer's Setup recipe configures the server as an administrator. The other server instances belong only to the PHP App Server layer, so they are configured as ordinary servers.

**Note**

The Custom Layers section used to include a walkthrough of a Redis custom layer. You can now use Amazon ElastiCache for Redis to provide Redis support to your app servers. For more information, see [Using ElastiCache Redis as an In-Memory Key-Value Store \(p. 305\)](#).

# Other Layers

AWS OpsWorks also supports the Ganglia and Memcached layers.

## Topics

- [Ganglia Layer \(p. 98\)](#)
- [Memcached \(p. 100\)](#)

## Ganglia Layer

AWS OpsWorks sends all of your instance and volume metrics to [Amazon CloudWatch](#), making it easy to view graphs and set alarms to help you troubleshoot and take automated action based on the state of your resources. You can also use the Ganglia AWS OpsWorks layer for additional application monitoring options such as storing your chosen metrics.

The Ganglia layer is a blueprint for an instance that monitors your stack by using [Ganglia](#) distributed monitoring. A stack usually has only one Ganglia instance. The Ganglia layer includes the following optional configuration settings:

### Ganglia URL

The statistic's URL path. The complete URL is `http://DNSName/URLPath`, where *DNSName* is the associated instance's DNS name. The default *URLPath* value is "/ganglia" which corresponds to something like: `http://ec2-54-245-151-7.us-west-2.compute.amazonaws.com/ganglia`.

### Ganglia user name

A user name for the statistics web page. You must provide the user name when you view the page. The default value is "opsworks".

### Ganglia password

A password that controls access to the statistics web page. You must provide the password when you view the page. The default value is a randomly generated string.

### Custom security groups

This setting appears if you chose to not automatically associate a built-in AWS OpsWorks security group with your layers. You must specify which security group to associate with the layer. For more information, see [Create a New Stack \(p. 41\)](#).

### Elastic Load Balancer

You can attach an Elastic Load Balancing load balancer to the layer's instances.

## Add Layer

OpsWorks    RDS

Layer type: Ganglia Looking for a different Layer type? [Let us know](#).

Ganglia Monitoring Master layer is a blueprint for an instance that monitors your stack and displays the results on a web page. [Learn more](#).

Ganglia URL: /ganglia

Ganglia user name: opsworks

Ganglia password: z4g60e447s

Elastic Load Balancer: No ELBs have been created in us-east-1. To add an ELB go to the [EC2 console](#).

Cancel    Add Layer

**Note**

Record the password for later use; AWS OpsWorks does not allow you to view the password after you create the layer. However, you can reset the password by going to the layer's Edit page and clicking **Reset now**.

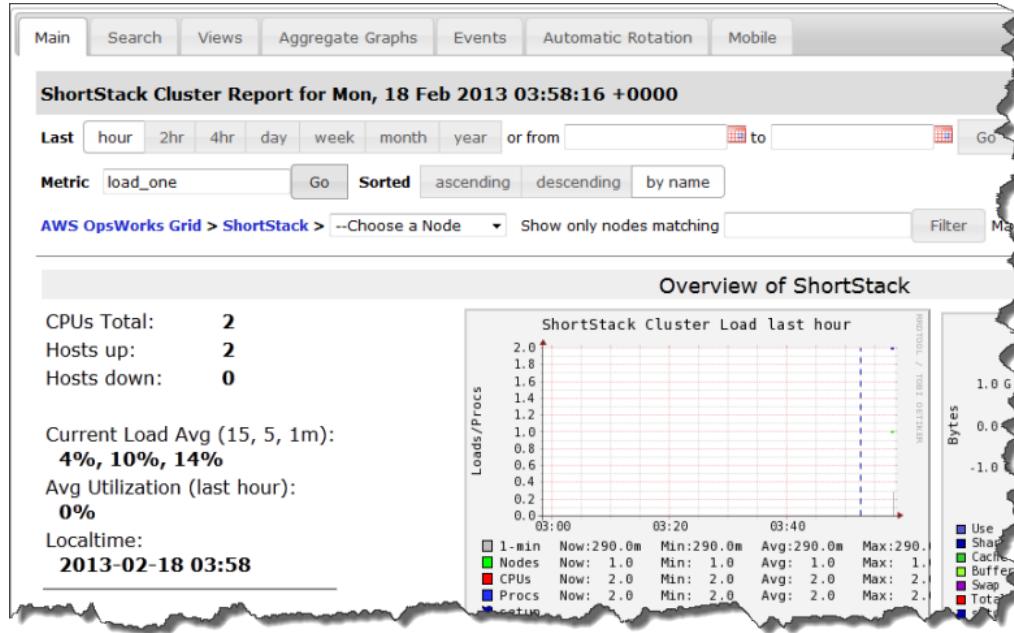
The screenshot shows the 'Layer Ganglia' settings page. It includes fields for 'Ganglia URL' (set to '/ganglia'), 'Ganglia username' (set to 'opsworks'), and 'Ganglia password' (with a note: 'Your password has already been set. [Reset now](#)'). Below these are sections for 'Built-in Chef recipes' (listing 17 defined recipes) and a link to 'Show >'.

## View the Ganglia Statistics

The standard AWS OpsWorks recipes install a low-overhead Ganglia client on every instance. If your stack includes a Ganglia layer, the Ganglia client automatically starts reporting to the Ganglia as soon as the instance comes on line. The Ganglia uses the client data to compute a variety of statistics and displays the results graphically on its statistics web page.

### To view Ganglia statistics

1. On the **Layers** page, click Ganglia to open the layer's details page.
2. In the navigation pane, click **Instances**. Under **Ganglia**, click the instance name.
3. Copy the instance's **Public DNS** name.
4. Use the DNS name to construct the statistics URL, which will look something like: <http://ec2-54-245-151-7.us-west-2.compute.amazonaws.com/ganglia>.
5. Paste the complete URL into your browser, navigate to the page, and enter the Ganglia user name and password to display the page. An example follows.



## Memcached

A Memcached layer is an AWS OpsWorks layer that provides a blueprint for instances that function as [Memcached](#) servers—a distributed memory-caching system for arbitrary data. The Memcached layer includes the following configuration settings.

### Allocated memory (MB)

(Optional) The amount of cache memory (in MB) for each of the layer's instances. The default is 512 MB.

### Custom security groups

This setting appears if you chose to not automatically associate a built-in AWS OpsWorks security group with your layers. You must specify which security group to associate with the layer. For more information, see [Create a New Stack \(p. 41\)](#).

## Add Layer

The dialog shows the following fields:

- Layer type:** Memcached (selected from a dropdown).
- Allocated memory (MB):** 512 (input field).
- Buttons:** Cancel and Add Layer.

Below the input fields, there is descriptive text: "A Memcached layer is a blueprint for instances that function as Memcached servers—a distributed memory-caching system for arbitrary data. [Learn more](#)".

# Instances

---

Instances represent the EC2 instances that handle the work of serving applications, balancing traffic, and so on. This section describes how to use AWS OpsWorks to manage a layer's instances.

## Topics

- [Operating Systems \(p. 101\)](#)
- [Adding an Instance to a Layer \(p. 103\)](#)
- [Using Custom AMIs \(p. 107\)](#)
- [Manually Starting, Stopping, and Rebooting 24/7 Instances \(p. 109\)](#)
- [Changing Instance Properties \(p. 111\)](#)
- [Deleting Instances \(p. 112\)](#)
- [Managing Load with Time-based and Load-based Instances \(p. 113\)](#)
- [Using SSH to Communicate with an Instance \(p. 119\)](#)

## Operating Systems

AWS OpsWorks ensures that instances have the latest security patches by calling `yum update` or `apt-get update` after an instance boots. For more control over how OpsWorks updates your instances, you can create custom AMIs. With this approach, you can, for example, specify which package versions are installed on an instance. For more information, see [Using Custom AMIs \(p. 107\)](#).

### Note

To avoid service interruptions, AWS OpsWorks does not install updates after the instance is online. However, you can manually install updates on online instances at any time by running the `update_dependencies` stack command. For more information, see [Run Stack Commands \(p. 52\)](#).

AWS OpsWorks supports the following built-in operating systems—[Amazon Linux](#) and [Ubuntu 12.04 LTS](#)—along with custom AMIs based on those operating systems. Each distribution has different support timelines and package-merge policies, so you should consider which approach best suits your requirements.

- [Amazon Linux](#)
- [Ubuntu 12.04 LTS](#)
- [Ubuntu 14.04 LTS](#)

You can also create custom AMIs based on those operating systems. Each distribution has different support timelines and package-merge policies, so you should consider which approach best suits your requirements. For more information about creating custom AMIs, see [Using Custom AMIs \(p. 107\)](#).

## Amazon Linux

By default, AWS OpsWorks installs all updates on boot, which provides you with the latest Amazon Linux release. In addition, Amazon Linux releases a new version approximately every six months, which can involve significant changes. For example, the 2012.09 to 2013.03 update upgraded the kernel from 3.2 to 3.4.

After a new Amazon Linux version is released, you can update your instances to the new version or stick with the old version, which will continue to receive updates for a two-week migration period. After that period ends, you can continue to use the old version by using lock-on-launch, which you enable by editing `/etc/yum.conf`, but you do not receive further updates. For more information, see [Amazon Linux AMI FAQs](#).

AWS OpsWorks handles a new Amazon Linux version by pausing automatic updates during the two-week migration period to give you time to move to the new version. For example, here's what happens during the two-week migration from Amazon Linux 2013.09 to 2014.03:

- Online instances that are running 2013.09 continue to run 2013.09.
- You can manually update an online 2013.09 instance [by running update\\_dependencies \(p. 52\)](#); AWS OpsWorks updates it to 2014.03 and disables lock-on-launch.
- When you start an existing stopped 2013.09 instance—including load-based and time-based instances—it will run 2013.09, unless you manually update it.
- When you create a new instance, it will use 2013.09.

A message in the console informs you that the instance will be using 2013.09. After the instance is online, you can update it to 2014.03 by running `update_dependencies`.

### Note

You can also update an instance from 2013.09 to 2014.03 by editing `etc/yum.conf` and changing `releasever=2013.09` to `releasever=2014.03`. For more information, see [How do I lock my AMI to a specific version?](#).

After the two-week migration period ends (on May 19 for the 2014.03 release):

- The AWS OpsWorks console displays a message informing you that all new instances will use Amazon Linux 2014.03.
- All new instances run 2014.03.
- Online 2013.09 instances—including load-based and time-based instances—continue to run 2013.09 until they are manually updated, but receive no further security updates.
- Stopped 2013.09 instance store-backed instances—including load-based and time-based instances—are automatically updated to 2014.03 when they are restarted.
- Stopped 2013.09 EBS-backed instances—including load-based and time-based instances—continue to run 2013.09 when they are restarted until they are manually updated, but receive no further security updates.

After an Amazon Linux version update is released, we recommend that you update your instances to the new version within two weeks, so your instances continue to receive security updates. Before updating your production stack's instances, we recommend you start a new instance and verify that your app runs correctly on the new version. You can then update the production stack to the new version by running `update_dependencies` on all instances.

### Important

Before running `update_dependencies` on t1.micro instances, make sure they have a temporary swap file, `/var/swapfile`. The t1.micro instances on Chef 0.9 stacks do not have a swap file. For Chef 11.4 and Chef 11.10 stacks, recent versions of the instance agent automatically create a swap file for t1.micro instances. However, this change was introduced over a period of several weeks, so you should check for the existence of `/var/swapfile` on instances created before approximately Mar. 24, 2014.

For t1.micro instances that lack a swap file, you can create one as follows:

- For Chef 11.4 and 11.10 stacks, create new t1.micro instances, which automatically have a swap file.
- For Chef 0.9 stacks, run the following commands on each instance as root user.

```
dd if=/dev/zero of=/var/swapfile bs=1M count=256
mkswap /var/swapfile
chown root:root /var/swapfile
chmod 0600 /var/swapfile
swapon /var/swapfile
```

You can also use these commands on Chef 11.4 or 11.10 stacks if you don't want to create new instances.

## Ubuntu LTS

Ubuntu releases a new Ubuntu LTS version approximately every two years and supports each release for approximately five years. Like Amazon Linux, Ubuntu provides security patches and updates for the duration of the operating system support. For more information, see [LTS - Ubuntu Wiki](#).

AWS OpsWorks supports two Ubuntu LTS versions, 12.04 and 14.04.

- Ubuntu 12.04 is supported for all stacks.
- Ubuntu 14.04 is supported only for Chef 11.10 stacks.

For more information, see [Chef Versions \(p. 161\)](#).

- You cannot update an existing Ubuntu 12.04 instance to Ubuntu 14.04.

You must [create a new Ubuntu 14.04 instance \(p. 103\)](#) and [delete the Ubuntu 12.04 instance \(p. 112\)](#).

## Adding an Instance to a Layer

After you create a layer, you usually add at least one instance. You can add more instances later, if the current set can't handle the load. You can also use load-based or time-based instances to automatically scale the number of instances. For more information, see [Managing Load with Time-based and Load-based Instances \(p. 113\)](#).

You can add either new or existing instances to a layer:

- **New**—OpsWorks creates a new instance, configured to your specifications, and makes it a member of the layer.
- **Existing**—You can add an existing instance from any compatible layer, but it must be in the offline (stopped) state.

If an instance belongs to multiple layers, AWS OpsWorks runs the recipes for each of the instance's layers when a lifecycle event occurs, or when you run a [stack \(p. 52\)](#) or [deployment \(p. 132\)](#) command. However, an instance's layers must be compatible with one another. For example, the HAProxy layer is not compatible with the Static Web Server layer because they both bind to port 80. An instance can be a member of only one of those layers. For more information, see [Appendix A: AWS OpsWorks Layer Reference \(p. 354\)](#).

**Note**

You can also make an instance a member of multiple layers by editing its configuration. For more information, see [Changing Instance Properties \(p. 111\)](#).

Each online instance has a `/etc/hosts` file that maps IP addresses to host names. AWS OpsWorks includes the public and private addresses for all of the stack's online instances in each instance's `hosts` file. For example, suppose that you have a stack with two Node.js App Server instances, `nodejs-app1` and `nodejs-app2`, and one MySQL instance, `db-master1`. The `nodejs-app1` instance's `hosts` file will look something like the following example, and the other instances' will have similar `hosts` files.

```
...
# OpsWorks Layer State
192.0.2.0 nodejs-app1.localdomain nodejs-app1
10.145.160.232 db-master1
198.51.100.0 db-master1-ext
10.243.77.78 nodejs-app2
203.0.113.0 nodejs-app2-ext
10.84.66.6 nodejs-app1
192.0.2.0 nodejs-app1-ext
```

## To add a new instance to a layer

1. On the **Instances** page, click **+Instance** for the appropriate layer and (if necessary) click the **New** tab. If you want to configure more than just the **Host name**, **Size**, and **Availability Zone**, click **Advanced >>** to see more options. The following shows the complete set of options:

PHP App Server

No instances. [Add an instance](#).

New Existing

Hostname	php-app1
Size <small>NEW DEFAULT</small>	c3.large
Availability Zone	us-east-1a
Scaling type	<input checked="" type="radio"/> 24/7 <input type="radio"/> Time-based <input type="radio"/> Load-based
SSH key	Do not use a default SSH key
Operating system <small>NEW</small>	Amazon Linux
Architecture	<input checked="" type="radio"/> 64bit <input type="radio"/> 32bit
Root device type	<input checked="" type="radio"/> Instance store <input type="radio"/> EBS backed

[Cancel](#) [Add Instance](#)

2. If desired, you can override the default configuration, most of which you specified when you created the stack. For more information, see [Create a New Stack \(p. 41\)](#).

#### Hostname

Identifies the instance on the network. By default, AWS OpsWorks generates each instance's host name by using the **Hostname theme** you specified when you created the stack. You can override this value and specify your preferred host name.

#### Size

An Amazon EC2 instance type, which specifies the instance's resources, such as the amount of memory or number of virtual cores. AWS OpsWorks specifies a default size for each instance, which you can override with your preferred instance type. AWS OpsWorks supports all instance types except Cluster Compute, Cluster GPU, and High Memory Cluster. For more information, see [Instance Families and Types](#).

#### Note

After the initial boot, Amazon EBS-backed instances boot faster than instance store-backed instances because AWS OpsWorks does not have to reinstall the instance's software from scratch. For more information, see [Storage for the Root Device](#)

#### Availability Zone/Subnet

If the stack is not in a VPC, this setting is labeled **Availability Zone** and lists the region's zones. Here you can override the default Availability Zone you specified when you created the stack and launch the instance in a different Availability Zone.

If the stack is in a VPC, this setting is labeled **Subnet** and lists VPC's subnets. Here you can override the default subnet you specified when you created the stack and launch the instance in a different subnet.

#### Tip

By default, AWS OpsWorks lists the subnet's CIDR ranges. To make the list more readable, use the VPC console or API to add a tag to each subnet with **Key** set to **Name** and **Value** set to the subnet's name. AWS OpsWorks appends that name to the CIDR range. In the preceding example, the subnet's Name tag is set to **Private**. For more information, see [Running a Stack in a VPC \(p. 44\)](#).

#### Scaling Type

Determines how the instance is started and stopped.

- The default value is a **24/7** instance, which you start and stop manually.
- AWS OpsWorks starts and stops **time-based** instances based on a specified schedule.
- AWS OpsWorks starts and stops **load-based** instances based on specified load metrics.

#### Note

You do not start or stop load-based or time-based instances yourself. Instead, you configure the instances, and AWS OpsWorks starts and stops them based on the configuration. For more information, see [Managing Load with Time-based and Load-based Instances \(p. 113\)](#).

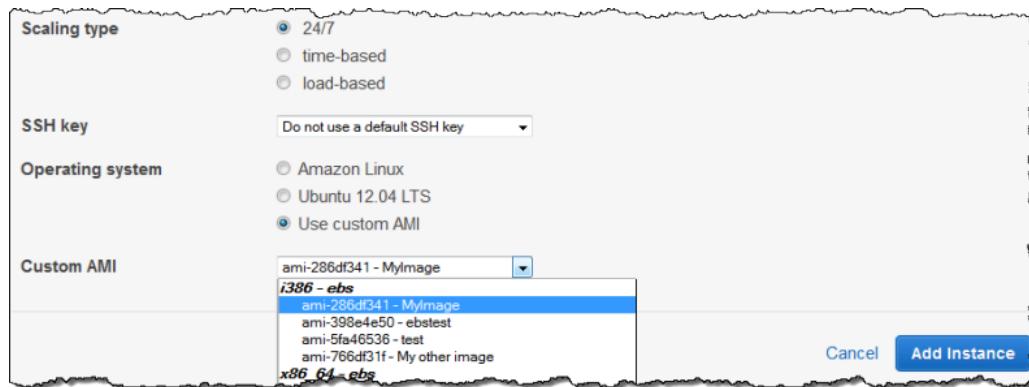
#### SSH key

An Amazon EC2 key pair. AWS OpsWorks installs the public key on the instance and you can use the private key with an SSH client to log into the instance. For more information, see [Using SSH to Communicate with an Instance \(p. 119\)](#). This field is initially set to the **Default SSH key** value that you specified when you created the stack.

- If the default value is set to **Do not use a default SSH key**, you can specify one of your account's Amazon EC2 keys.
- If the default value is set to an Amazon EC2 key, you can specify a different key or no key.

#### Operating system

**Operating system** specifies which operating system the instance is running. Initially, this field is set to the **Default operating system** value that you specified when you created the stack. You can override that value to specify a different operating system. If you select **Use custom AMI**, the page displays a list of custom AMIs instead of **Architecture** and **Root device type**.



For more information on how to use custom AMIs, see [Using Custom AMIs \(p. 107\)](#).

#### Architecture

Specifies whether the instance's CPU is 32-bit or 64-bit. The default value is 64-bit but you can also specify 32-bit.

#### Root device type

Determines the type of storage available to the instance. The default root device is the instance store, but you can also specify an Amazon EBS-backed root device. For more information, see [Storage](#).

3. Click **Add Instance** to create the new instance.

### To add an existing instance to a layer

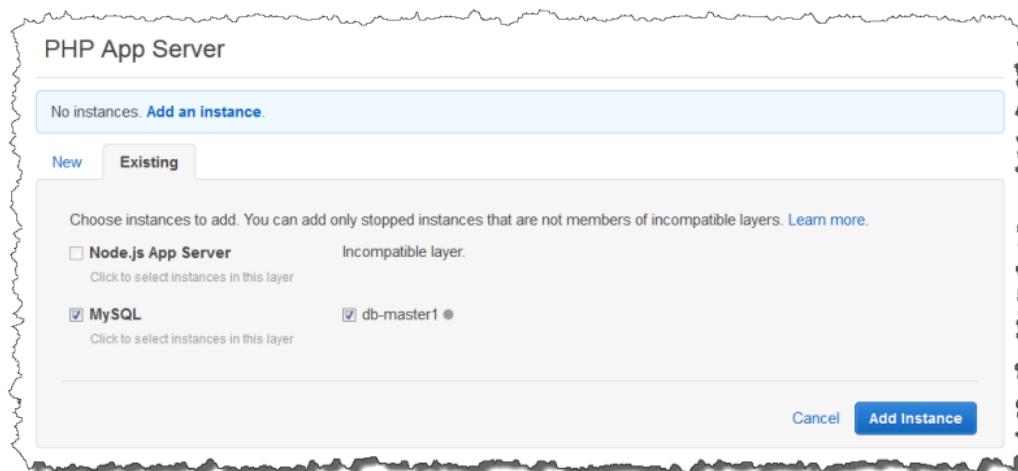
1. On the **Instances** page, click **+Instance** for the appropriate layer and then click the **Existing** tab.

#### Note

If you change your mind about using an existing instance, click **New** to create a new instance as described in the preceding procedure.

2. On the **Existing** tab, select an instance from the list. The list shows all instances, but lets you select only offline instances from compatible layers. For this example, the db-master1 instance will belong to the PHP App Server and MySQL layers, and will run both servers.

If an instance belongs to an incompatible layer, it won't be available here, even if it also belongs to a compatible layer. For example, the PHP App Server and Node.js App Server layers are compatible with MySQL but not with each other. Suppose that your stack has a MySQL layer with two instances, db-master1 and db-master2. If you make db-master2 a member of the Node.js App Server layer, it cannot be also be a member of the PHP App Server layer. If you try to add an existing instance to a PHP App Server layer, the **Add Instance** section will show only db-master1.



3. Click **Add Instance** to create the new instance.

An instance represents an Amazon EC2 instance, but is basically just an AWS OpsWorks data structure. An instance must be started to create a running Amazon EC2 instance, as described in the following sections.

**Important**

If you launch instances into a default VPC, you must be careful about modifying the VPC configuration. The instances must always be able to communicate with the AWS OpsWorks service, Amazon S3, and the package repositories. If, for example, you remove a default gateway, the instances will lose their connection to the AWS OpsWorks service, which will then treat the instances as failed and [auto heal \(p. 67\)](#) them. However, AWS OpsWorks will not be able to install the instance agent on the healed instances. Without an agent, the instances cannot communicate with the service, and the startup process will not progress beyond the "booting" status. For more information on default VPC, see [Supported Platforms](#).

## Using Custom AMIs

AWS OpsWorks supports two ways to customize instances: custom Amazon Machine Images (AMIs) and custom Chef recipes. Both approaches give you control over which packages and package versions are installed, how they are configured, and so on. However, each approach has different advantages, so the best one depends on your requirements.

The following are the primary reasons to consider using a custom AMI:

- You want to pre-bundle specific packages instead of installing them after the instance boots.
- You want to control the timing of package updates to provide a consistent base image for your layer.
- You want instances—[load-based \(p. 113\)](#) instances in particular—to boot as quickly as possible.

Chef recipes are more flexible than custom AMIs, easier to update, and can perform updates on running instances. In practice, the optimal solution might be a combination of both approaches. For more information on custom recipes, see [Cookbooks and Recipes \(p. 153\)](#) and [Customizing AWS OpsWorks \(p. 230\)](#).

**Note**

You cannot use a custom AMI as a stack's default operating system. You can specify custom AMIs only when you add new instances to layers. For more information, see [Adding an Instance to a Layer \(p. 103\)](#) and [Create a New Stack \(p. 41\)](#).

To use a custom AMI with AWS OpsWorks, you must first create an AMI from a customized instance. You can choose from two basic approaches:

- Use the Amazon EC2 console or API to create and customize an instance, based on one of the AWS OpsWorks-supported AMIs: [Amazon Linux](#), [Ubuntu 12.04 LTS](#), and [Ubuntu 14.04 LTS](#).
- Use OpsWorks to create an Amazon EC2 instance, based on the configuration of its associated layers.

**Tip**

Be aware that an AMI might not work with all instance types, so make sure that your starting AMI is compatible with the instance types that you plan to use. In particular, the [R3](#) instance types require a hardware-assisted virtualization (HVM) AMI.

You then use the Amazon EC2 console or API to create a custom AMI from the customized instance. You can use your custom AMIs in any stack that is in the same region by adding an instance to a layer and specifying your custom AMI. For more information on how to create an instance that uses a custom AMI, see [Adding an Instance to a Layer \(p. 103\)](#).

**Note**

By default, AWS OpsWorks installs all Amazon Linux updates on boot, which provides you with the latest release. In addition, Amazon Linux releases a new version approximately every six months, which can involve significant changes. To defer updating to a new version, you can lock your custom AMI to a specified version. For more information on how to lock an AMI, see the [Amazon Linux FAQ](#). For more information about Amazon Linux and AWS OpsWorks, see [Operating Systems \(p. 101\)](#).

The simplest way to create a custom AMI is to perform the entire task by using the Amazon EC2 console or API. For more details about the following steps, see [Creating Your Own AMIs](#).

### To create a custom AMI using Amazon EC2 console or API

1. Create an instance by using one of the following AMIs: [Amazon Linux](#), [Ubuntu 12.04 LTS](#), or [Ubuntu 14.04 LTS](#).
2. Customize the instance from Step 1 by configuring it, installing packages, and so on. Remember that everything you install will be reproduced on every instance based on the AMI, so don't include items that should be specific to a particular instance.
3. Stop the instance and create a custom AMI.

If you want to use a customized AWS OpsWorks instance to create an AMI, you should be aware that every Amazon EC2 instance created by OpsWorks includes a unique identity. If you create a custom AMI from such an instance, it will include that identity and all instances based on the AMI will have the same identity. To ensure that the instances based on your custom AMI have a unique identity, you must remove the identity from the customized instance before creating the AMI.

### To create a custom AMI from an AWS OpsWorks instance

1. [Create a stack \(p. 41\)](#) and [add one or more layers \(p. 58\)](#) to define the configuration of the customized instance. You can use built-in layers, customized as appropriate, as well as fully custom layers. For more information, see [Customizing AWS OpsWorks \(p. 230\)](#).
2. [Edit the layers \(p. 59\)](#) and disable AutoHealing.
3. [Add an instance \(p. 103\)](#) to the layer or layers and [start it \(p. 109\)](#). We recommend using an Amazon EBS-backed instance. Open the instance's details page and record its Amazon EC2 ID for later.
4. When the instance is online, connect to it with SSH and run the following commands, in order:
  1. `sudo /etc/init.d/monit stop`
  2. `sudo /etc/init.d/opsworks-agent stop`

3. `sudo rm -rf /etc/aws/opsworks/ /opt/aws/opsworks/ /var/log/aws/opsworks/ /var/lib/aws/opsworks/ /etc/monit.d/opsworks-agent.monitrc /etc/monit/conf.d/opsworks-agent.monitrc /var/lib/cloud/`

5. This step depends on the instance type:

- For an Amazon EBS-backed instance, use the AWS OpsWorks console to [stop the instance \(p. 109\)](#) and create the AMI as described in [Creating an Amazon EBS-Backed Linux AMI..](#)
- For an instance store-backed instance, create the AMI as described in [Creating an Instance Store-Backed Linux AMI](#) and then use the AWS OpsWorks console to stop the instance.

When you create the AMI, be sure to include the certificate files. For example, you can call the `ec2-bundle-vol` command with the `-i` argument set to `-i $(find /etc /usr /opt -name '*.*.pem' -o -name '*.*.crt' -o -name '*.*.gpg' | tr '\n' ',')`. Do not remove the apt public keys when bundling. The default `ec2-bundle-vol` command handles this task.

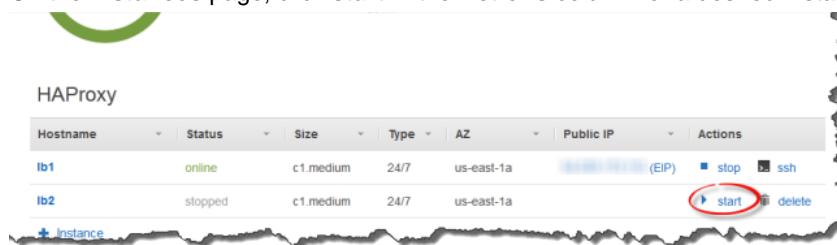
6. Clean up your stack by returning to the AWS OpsWorks console and [deleting the instance \(p. 112\)](#) from the stack.

## Manually Starting, Stopping, and Rebooting 24/7 Instances

You must manually start a 24/7 instance to create the corresponding Amazon EC2 instance and manually stop it to terminate the instance. You can also manually reboot instances that are not functioning properly. AWS OpsWorks automatically starts and stops time-based and load-based instances. For more information, see [Managing Load with Time-based and Load-based Instances \(p. 113\)](#).

### To start an instance

- On the **Instances** page, click **start** in the **Actions** column for a desired instance.



You can also create multiple instances and then start them all at the same time by clicking **Start all Instances**.

After you click **start**, AWS OpsWorks creates an EC2 instance and boots the operating system. It typically takes a few minutes to boot the Amazon EC2 instance and install the packages. As startup progresses, the instance's **Status** field displays the following series of values:

1. **requested** - AWS OpsWorks has called the Amazon EC2 service to create the Amazon EC2 instance.
2. **pending** - AWS OpsWorks is waiting for the Amazon EC2 instance to start.
3. **booting** - The Amazon EC2 instance is booting.

4. **running\_setup** - AWS OpsWorks is running the layer's Setup recipes, followed by the Deploy recipes. For more information, see [Executing Recipes \(p. 175\)](#). If you have added [added custom cookbooks \(p. 171\)](#), to the stack, AWS OpsWorks installs the current version.
5. **online** - The instance is ready for use.

**Note**

AWS OpsWorks does not necessarily deploy the latest cookbook and app versions to restarted offline instances. For more information, see [Editing Apps \(p. 134\)](#) and [Updating Custom Cookbooks \(p. 174\)](#).

When the **Status** changes to **online**, the instance is fully operational. AWS OpsWorks replaces the **start** action with **stop**, which you can use to terminate the associated Amazon EC2 instance. A stopped instance is still part of the stack and retains all resources. For example, Amazon EBS volumes and Elastic IP addresses are still attached to a stopped instance. If you start the instance again, AWS OpsWorks creates a new EC2 instance and attaches those resources to it.

If the instance did not start successfully, or the setup recipes failed, the status will be set to `start_failed` or `setup_failed`, respectively. You can examine the logs to determine the cause. For more information, see [Debugging and Troubleshooting Guide \(p. 337\)](#).

Stopping an instance terminates the associated Amazon EC2 instance. This action deletes the data on instance store-backed instances, but not on Amazon EBS-backed instances.

### To stop an instance

1. On the **Instance** page, click **stop** for an instance, which notifies AWS OpsWorks to run the shutdown recipes and terminate the EC2 instance.
2. Click **Yes, stop** to confirm.

## Instances

Stop All Instances

An instance represents an EC2 instance. Each instance belongs to a layer that defines the instance's settings, resources, installed packages, profiles and security groups. When you start the instance, OpsWorks uses the associated layer's blueprint to create and configure a corresponding EC2 instance. [Learn more](#).

1  
Instances

1online

0launching

0shutting down

0stopped

0error

### PHP App Server

Host Name	Status	Size	Type	AZ	Public IP	Actions
php-app1	online	c1.medium	24/7	us-east-1a	54.242.127.207	

Are you sure you want to stop php-app1?

All data not stored on EBS volumes will be lost.

CancelStop

+ Instance?

You can also shut down every instance in the stack by clicking **Stop All Instances**. You can monitor the shutdown process by watching the **Status** column.

As shutdown progresses, the instance's **Status** field displays the following series of values:

1. **terminating** - AWS OpsWorks is terminating the Amazon EC2 instance.
2. **shutting\_down** - AWS OpsWorks is running the layer's Shutdown recipes.
3. **terminated** - The Amazon EC2 instance is terminated.
4. **stopped** - The instance has stopped.

Rebooting restarts the associated Amazon EC2 instance but does not delete the instance's data.

### To reboot an instance

1. On the **Instances** page, click the nonfunctioning instance to open the details page.
2. Click **Reboot**.



#### Tip

You can have AWS OpsWorks automatically replace failed instances by enabling auto healing. For more information, see [How to Use Auto Healing to Replace a Layer's Failed Instances \(p. 67\)](#).

## Changing Instance Properties

Some instance properties, such as **Availability Zone** and **Scaling Type**, can be set only when you create an instance. Other properties can be modified later, but only when the instance is stopped. After an instance is online, you can't modify its configuration directly but you can change some aspects of the configuration by editing the instance's layers. For more information, see [How to Edit an OpsWorks Layer \(p. 59\)](#).

### To modify instance configuration

1. Stop the instance, if it is not already stopped.
2. On the **Instances** page, click an instance name to display the **Details** page.
3. Click **Edit** to display the edit page.
4. Modify the instance's configuration, as appropriate.

For a description of the **Host name**, **Size**, **SSH key** and **Operating system** settings, see [Adding an Instance to a Layer \(p. 103\)](#). The **Layers** setting lets you add or remove layers. The instance's current layer's appear following the list of layers.

- To add another layer, select it from the list.
- To remove the instance from one of its layers, click the **x** by the appropriate layer.

An instance must be a member of at least one layer, so you cannot remove the last layer.

When you restart the instance, AWS OpsWorks starts a new Amazon EC2 instance with the updated configuration.

# Deleting Instances

Stopping an instance terminates the EC2 instance, but the instance remains in the stack. You can restart it by clicking **start**, and AWS OpsWorks creates a new EC2 instance. If you no longer need an instance, you can delete it.

## Note

This topic applies only to instances that were created by using AWS OpsWorks. For more information about how to delete instances that were created by using the Amazon EC2 console or API, see [Terminate Your Instance](#).

If an instance belongs to multiple layers, you can delete the instance from the stack or just remove a particular layer. You can also remove layers from instances by editing the instance configuration, as described in [Changing Instance Properties \(p. 111\)](#).

## Important

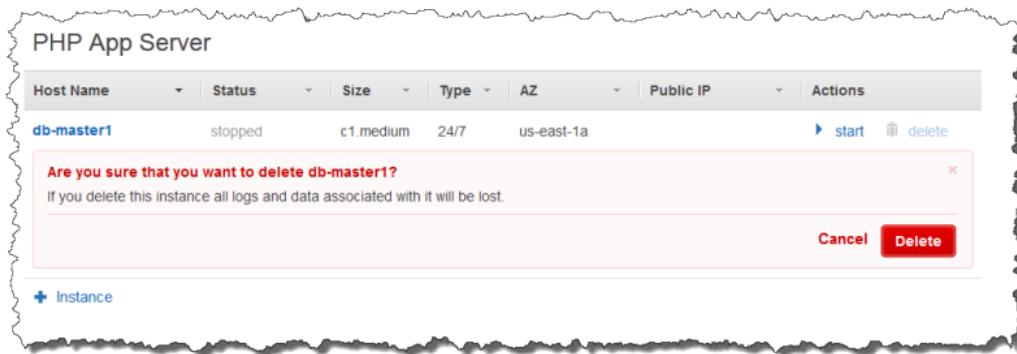
You should delete instances only by using the AWS OpsWorks console or API. In particular, do not delete AWS OpsWorks instances by using the Amazon EC2 console or API. Amazon EC2 actions are not automatically synchronized with AWS OpsWorks. For example, if auto healing is enabled and you terminate an instance by using Amazon EC2, AWS OpsWorks treats the terminated instance as a failed instance and starts another instance to replace it. For more information, see [How to Use Auto Healing to Replace a Layer's Failed Instances \(p. 67\)](#).

## To delete an instance

1. On the **Instances** page, locate the instance under the appropriate layer. If the instance is running, click **stop** in the **Actions** column.
2. After the status changes to **stopped**, click **delete**. If the instance is a member of more than one layer, AWS OpsWorks displays the following section.



- To remove the instance from only the selected layer, click **Remove from layer**.  
The instance remains a member of its other layers and can be restarted.
  - To delete the instance from all its layers, which removes it from the stack, click **here**.
3. If you choose to completely remove an instance from the stack, or if the instance is a member of only one layer, AWS OpsWorks displays the following section.



Click **Yes, delete** to confirm. In addition to deleting the instance from the stack, this action deletes any associated logs or data.

## Managing Load with Time-based and Load-based Instances

As your incoming traffic varies, your stack may have either too few instances to comfortably handle the load or more instances than necessary. You can save both time and money by using time-based or load-based instances to automatically increase or decrease a layer's instances so that you always have enough instances to adequately handle incoming traffic without paying for unneeded capacity. There's no need to monitor server loads or manually start or stop instances. In addition, time- and load-based instances automatically distribute, scale, and balance applications over multiple Availability Zones within a region, giving you geographic redundancy and scalability.

Automatic scaling is based on two instance types, which adjust a layer's online instances based on different criteria:

- **Load-based** instances allow your stack to handle variable loads by starting additional instances when traffic is high and stopping instances when traffic is low, based on any of several load metrics. For example, you can have AWS OpsWorks start instances when the average CPU utilization exceeds 80% and stop instances when the average CPU load falls below 60%.
- **Time-based** instances allow your stack to handle loads that follow a predictable pattern by including instances that run only at certain times or on certain days. For example, you could start some instances after 6PM to perform nightly backup tasks or stop some instances on weekends when traffic is lower.

Unlike 24/7 instances, which you must start and stop manually, you do not start or stop time-based or load-based instances yourself. Instead, you configure the instances and AWS OpsWorks starts or stops them based on their configuration. For example, you configure time-based instances to start and stop on a specified schedule. AWS OpsWorks then starts and stops the instances according to that configuration.

A common practice is to use all three instance types together, as follows.

- A set 24/7 instances to handle the base load. You typically just start these instances and let them run continuously.
- A set of time-based instances, which AWS OpsWorks starts and stops to handle predictable traffic variations. For example, if your traffic is highest during working hours, you would configure the time-based instances to start in the morning and shut down in the evening.
- A set of load-based instances, which AWS OpsWorks starts and stops to handle unpredictable traffic variations. AWS OpsWorks starts them when the load approaches the capacity of the stacks' 24/7 and time-based instances, and stops them when the traffic returns to normal..

**Note**

If you have created apps for the instances' layer or created custom cookbooks, AWS OpsWorks automatically deploys the latest version to time-based and load-based instances when they are first started. However, AWS OpsWorks does not necessarily deploy the latest cookbooks to re-started offline instances. For more information, see [Editing Apps \(p. 134\)](#) and [Updating Custom Cookbooks \(p. 174\)](#).

**Topics**

- [How Automatic Time-based Scaling Works \(p. 114\)](#)
- [How Automatic Load-based Scaling Works \(p. 116\)](#)
- [How Load-based Scaling Differs from Auto Healing \(p. 119\)](#)

## How Automatic Time-based Scaling Works

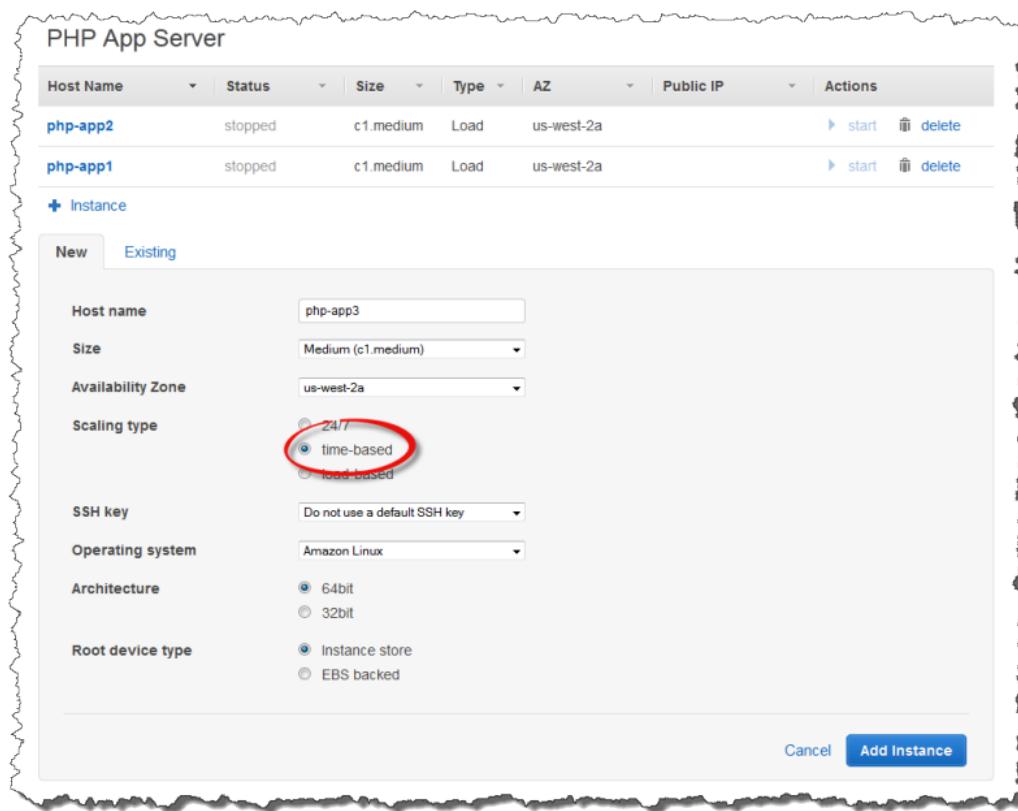
Time-based scaling lets you control how many instances a layer should have online at certain times of day or days of the week by starting or stopping instances on a specified schedule. AWS OpsWorks checks every couple of minutes and starts or stops instances as required. You specify the schedule separately for each instance, as follows:

- Time of day. You can have more instances running during the day than at night, for example.
- Day of the week. You can have more instances running on weekdays than weekends, for example.

You can either add a new time-based instance to a layer, or use an existing instance.

### To add a new time-based instance

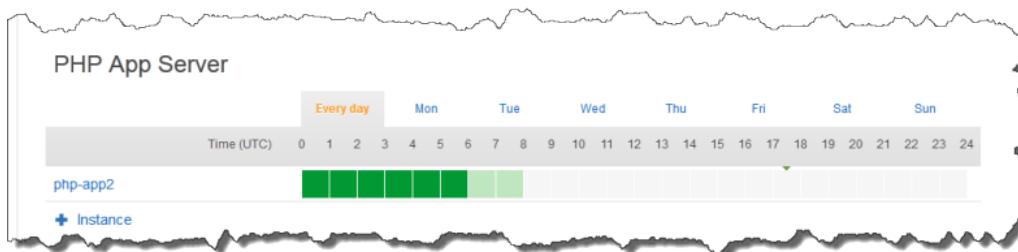
1. On the **Instances** page, click **+ Instance** to add an instance. On the **New** tab, click **Advanced >>** and then click **time-based**.



- Configure the instance as desired. Then click **Add Instance** to add the instance to the layer.

### To specify a schedule for a time-based instance

- In the navigation pane, click **Time-based** under **Instances**.
- Specify the online periods for each time-based instance by clicking the appropriate boxes below the desired hour.
  - To use the same schedule every day, click the **Every day** tab and then specify the online time periods.
  - To use different schedules on different days, click each day and select the appropriate time periods.



A light green square indicates that the instance is online on only some days. Dark green indicates that the instance is on line at this time every day. For example, the php-app2 instance is online some days from UTC 0000h to 0600h and every day from 0000h to 0800h.

**Note**

Make sure to allow for the amount of time it takes to start an instance and the fact that AWS OpsWorks checks only every few minutes to see if instances should be started or stopped. For example, if an instance should be running by 1:00 UTC, start it at 0:00 UTC. Otherwise, AWS OpsWorks might not start the instance until several minutes past 1:00 UTC, and it will take several more minutes for it to come online.

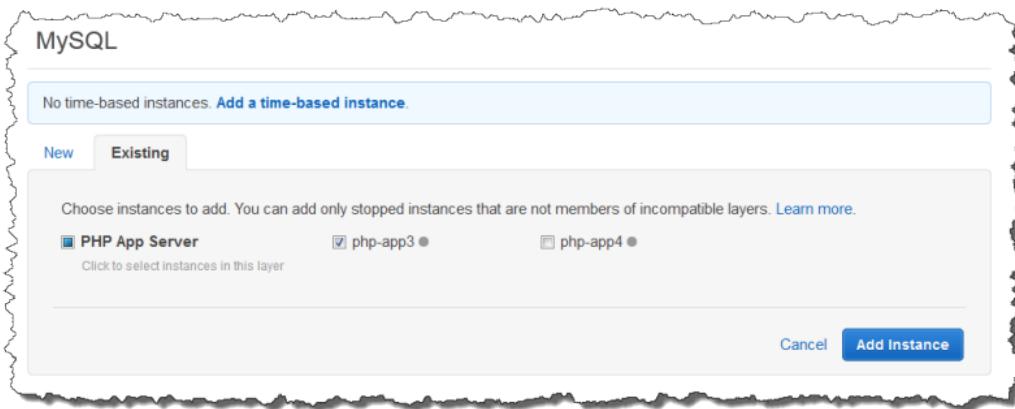
You can modify an instance's online time periods at any time using the previous configuration steps. The next time AWS OpsWorks checks, it uses the new schedule to determine whether to start or stop instances.

**Note**

You can also add a new time-based instance to a layer by going to the **Time-based** page and clicking **Add a time-based instance** (if you have not yet added a time-based instance to the layer) or **+Instance** (if the layer already has one or more time-based instances). Then configure the instance as described in the preceding procedures.

**To add an existing time-based instance to a layer**

1. On the **Time-based Instances** page, click **+ Instance** if a layer already has a time-based instance. Otherwise, click **Add a time-based instance**. Then click the **Existing** tab.



2. On the **Existing** tab, select an instance from the list. The list shows only time-based instances.

**Note**

If you change your mind about using an existing instance, click the **New** tab to create a new instance, as described in the preceding procedure.

3. Click **Add instance** to add the instance to the layer.

## How Automatic Load-based Scaling Works

Load-based instances let you rapidly start or stop instances in response to changes in incoming traffic. AWS OpsWorks uses [Amazon CloudWatch](#) data to compute the following metrics for each layer, which represent average values across all of the layer's instances:

- CPU: The average CPU consumption, such as 80%
- Memory: The average memory consumption, such as 60%
- Load: The average computational work a system performs in one minute.

You define *upscale* and *downscale* thresholds for any or all of these metrics and specify how AWS OpsWorks responds if the load exceeds a threshold. You can configure all of the following:

- How many instances to start or stop.

- How long AWS OpsWorks should wait after exceeding a threshold before starting or deleting instances. For example, CPU utilization must exceed the threshold for at least 15 minutes. This value allows you to ignore brief traffic fluctuations.
- How long AWS OpsWorks should wait after starting or stopping instances before monitoring metrics again. You usually want to allow enough time for started instances to come online or stopped instances to shut down before assessing whether the layer is still exceeding a threshold.

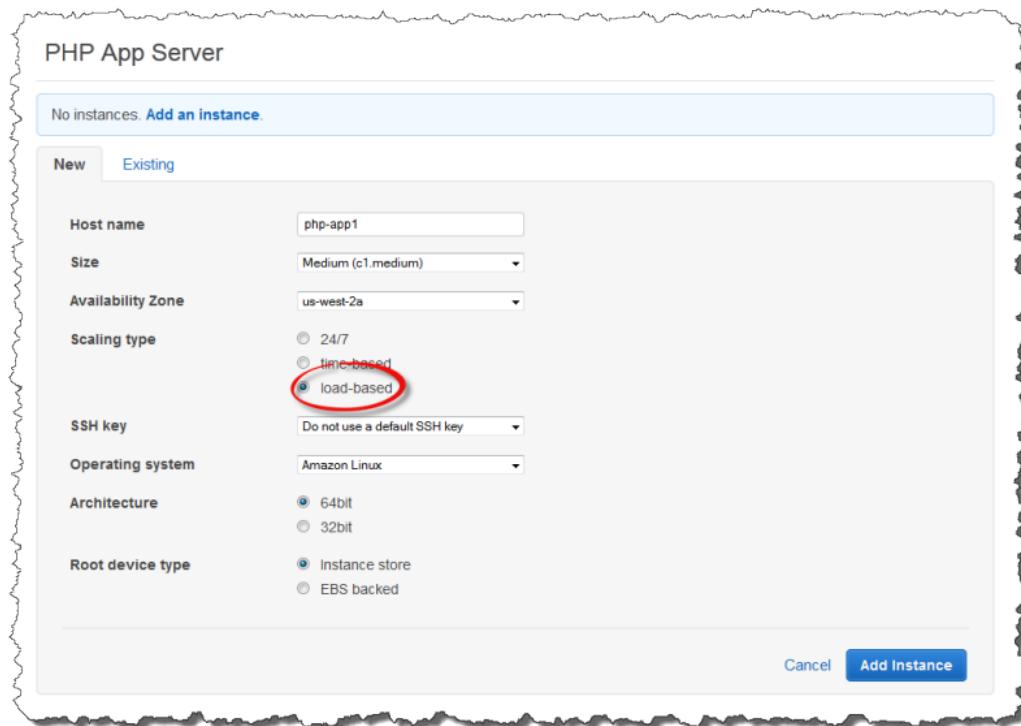
When you exceed a metric, AWS OpsWorks starts or stops only load-based instances. It does not start or stop 24/7 instances or time-based instances.

**Note**

Automatic load-based scaling does not create new instances; it starts and stops only those instances that you have created. You must therefore provision enough load-based instances in advance to handle the maximum anticipated load.

### To create a load-based instance

1. On the **Instances** page, click **+Instance** to add an instance. Click **Advanced >** and then click **load-based**.



2. Configure the instance as desired. Then click **Add Instance** to add the instance to the layer.

Repeat this procedure until you have created a sufficient number of instances. You can add or remove instances later, as required.

After you have added load-based instances to a layer, you must enable load-based scaling and specify the configuration. The load-based scaling configuration is a layer property, not an instance property, that specifies when a layer should start or stop its load-based instances. It must be specified separately for each layer that uses load-based instances.

## To enable and configure automatic load-based scaling

1. In the navigation pane, click **Load-based** under **Instances** and click **edit** for the appropriate layer.

2. Toggle **Load-based auto scaling enabled** to **Yes**. Then set your desired thresholds.

### Load-based instances

OpsWorks automatically starts and stops load-based instances in response to CPU, memory, and application load changes across all the instances in a layer. When a metric exceeds its Up threshold, OpsWorks starts more instances and when a metric falls below its Down threshold, OpsWorks stops some instances. [Learn more](#).

#### PHP App Server

Start/stop servers in batches of	If thresholds are exceeded/undershot for	After scaling up/down, ignore metrics for	Average CPU	or	Average memory	or	Average load
Up <input type="text" value="1"/>	<input type="text" value="5"/> min	<input type="text" value="5"/> min	<input type="text" value="80"/> %	or	<input type="text" value=""/> %	or	<input type="text" value=""/> %
Down <input type="text" value="1"/>	<input type="text" value="10"/> min	<input type="text" value="10"/> min	<input type="text" value="30"/> %	or	<input type="text" value=""/> %	or	<input type="text" value=""/> %
Load-based auto scaling <input checked="" type="checkbox"/> Yes							
0 of 1 instances are running <a href="#">show »</a>							
<a href="#">+ Instance</a>							

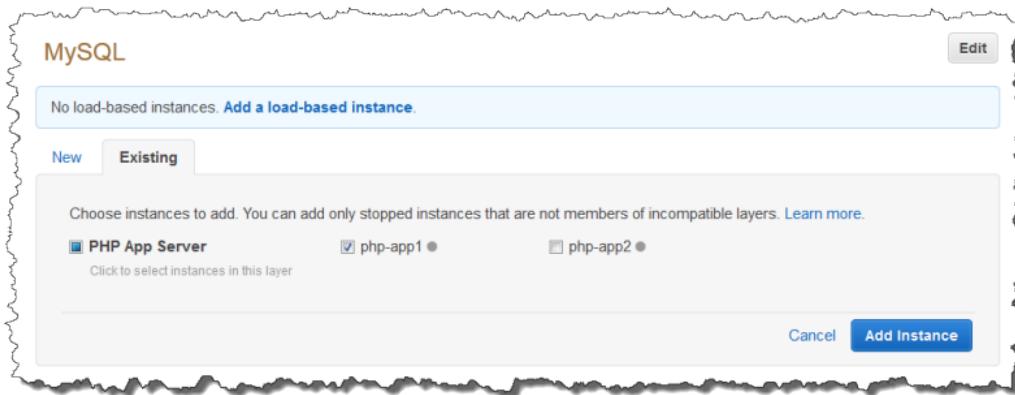
3. To add additional load-based instances, click **+ Instance**, configure the settings, and click **Add Instance**. Repeat until you have enough load-based instances to handle your maximum anticipated load. Then click **Save**.

#### Note

You can also add a new load-based instance to a layer by going to the **Load-based** page and clicking **Add a load-based instance** (if you have not yet added a load-based instance to the layer) or **+Instance** (if the layer already has one or more load-based instances). Then configure the instance as described earlier in this section.

## To add an existing load-based instance to a layer

1. In the navigation pane, click **Load-based** under **Instances**.
2. If you have already enabled load-based auto scaling for a layer, click **+Instance**. Otherwise, click **Add a load-based instance**. Then click the **Existing** tab.



3. On the **Existing** tab, select an instance from the list. The list shows only load-based instances.

**Note**

If you change your mind about using an existing instance, click the **New** tab to create a new instance as described in the preceding procedure.

4. Click **Add Instance** to add the instance to the layer.

You can modify the configuration for or disable automatic load-based scaling at any time.

#### To disable automatic load-based scaling

1. In the navigation pane, click **Load-based** under **Instances** and click **edit** for the appropriate layer.
2. Toggle **Load-based auto scaling enabled** to **No**.

## How Load-based Scaling Differs from Auto Healing

Automatic load-based scaling uses load metrics that are averaged across all running instances. If the metrics remain between the specified thresholds, AWS OpsWorks does not start or stop any instances. With auto healing, on the other hand, AWS OpsWorks automatically starts a new instance with the same configuration when an instance stops responding. The instance may not be able to respond due to a network issue or some problem with the instance.

For example, suppose that your CPU upscaling threshold is 80%, and then one instance stops responding.

- If auto healing is disabled, and the remaining running instances are able to keep average CPU utilization below 80%, AWS OpsWorks does not start a new instance. It starts a replacement instance only if the average CPU utilization across the remaining instances exceeds 80%.
- If auto healing is enabled, AWS OpsWorks starts a replacement instance irrespective of the load thresholds.

## Using SSH to Communicate with an Instance

You can use SSH to log into any of your online instances. There are two basic approaches:

- Use the built-in MindTerm SSH client. (p. 120)
- Use a third-party client. (p. 122)

# Using the MindTerm SSH Client

The simplest way to log into an instance is to use the MindTerm SSH client. The details depend on whether you are logged in as an IAM (AWS Identity and Access Management) user or are using your account's access and secret keys.

## Note

You must have Java enabled in your browser to use the MindTerm client.

If you log in to AWS OpsWorks as an IAM user, before using the MindTerm client for the first time, you must configure AWS OpsWorks.

## To use the MindTerm SSH client as an IAM user

1. In your browser, navigate to your IAM account alias. For more information, see [Managing User Permissions \(p. 277\)](#).
2. Log in as the IAM user that will use the client.
3. Create an SSH key pair for the user and give OpsWorks the public key. For details, see [Setting an IAM User's Public SSH Key \(p. 292\)](#).

Store the private key for later use.

4. In the OpsWorks navigation pane, click **Permissions**. Select the **SSH** checkbox for the desired IAM user to grant the necessary permissions. If you want to allow the user to use `sudo` to elevate privileges—for example, to run [agent CLI \(p. 368\)](#) commands—check the **sudo** box as well.

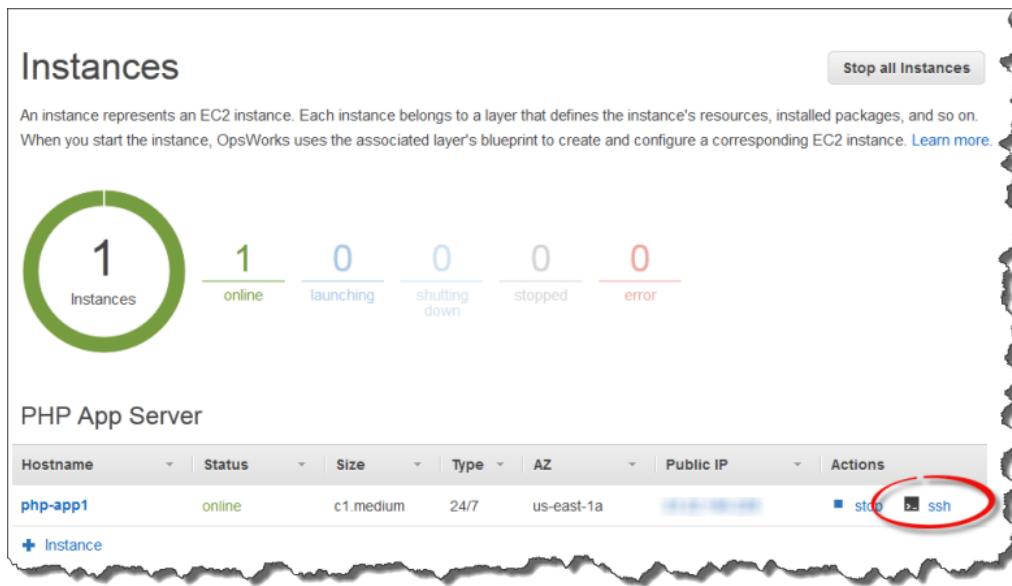
The screenshot shows the AWS OpsWorks console with the "Permissions" section selected in the navigation pane. The main area displays a table of IAM users with columns for "User name", "SSH", "sudo", and "Actions". A red circle highlights the "sudo" checkbox for the user "opsuser001", which is checked. The "Actions" column contains links for "Edit" and "Delete". At the bottom left of the table, there is a link "+ Import IAM Users".

User name	SSH	sudo	Actions
opsuser001	<input type="checkbox"/>	<input type="checkbox"/>	<a href="#">Edit</a> <a href="#">Delete</a>
opsuser002	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<a href="#">Edit</a> <a href="#">Delete</a>
opsuser003	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<a href="#">Edit</a> <a href="#">Delete</a>
opsuser004	<input type="checkbox"/>	<input type="checkbox"/>	<a href="#">Edit</a> <a href="#">Delete</a>

Each online instance includes an **ssh** action that you can use to open an SSH link to the instance by using the MindTerm client.

## To open an SSH link by using the MindTerm client

1. Log in as the IAM user whose permissions you set in the previous procedure.
2. On the **Instances** page, click **ssh** in the **Actions** column for the appropriate instance.



3. For **Path to your private key**, specify the location of the IAM user's private key, which must correspond to the public key from the previous procedure. Then click **Launch Mindterm**.

## SSH web1

Connect from your browser using the Java SSH Client (Java Required)

In order to launch MindTerm, you need to enter the path to the private key located on your hard drive. AWS automatically detects the key pair name and public DNS for your instance.

Path to your private key

**Launch Mindterm**

4. Use the terminal window to run commands on the instance.

The screenshot shows a terminal window titled '@web1:~ [80x24]' with a blue header bar and a light gray body. The window contains the following text:

```
MindTerm home: C:\Users\[REDACTED]\mindterm\  
Initializing random generator, please wait...done  
Connected to server running SSH-2.0-OpenSSH_5.3  
  
Server's hostkey (ssh-rsa) fingerprint:  
openssh md5: 7c:1a:61:b8:aa:68:d7:9d:bd:21:dd:a7:12:f3:11:51  
bubblebabble: xepaf-tuhor-komop-cytad-migod-ficyz-vipis-falyz-nisal-sokyl-naxax  
  
Last login: Fri Mar 15 19:19:06 2013 from 205.251.233.48  
This instance is managed with AWS OpsWorks.  
  
##### OpsWorks Summary #####  
Operating System: Amazon Linux AMI release 2012.09  
OpsWorks Instance: web1  
OpsWorks Instance ID: 7c43a8bd-61b5-47c1-b143-b0338908c7d8  
OpsWorks Layers: web  
OpsWorks Stack: APIStack  
EC2 Region: us-west-2  
EC2 Availability Zone: us-west-2a  
  
Visit http://aws.amazon.com/opsworks for more information.  
[ [REDACTED] ~]$
```

The procedure is similar for root users.

#### To open an SSH link using your access and secret keys

1. Log in using your account's access and secret keys.
2. Assign an Amazon EC2 SSH key to the instance. For more information, see [Using a Third-Party Client with Amazon EC2 Key Pairs \(p. 122\)](#).
3. On the **Instances** page, click **ssh** in the **Actions** column for the appropriate instance.
4. For **Path to your private key**, specify the location of the private Amazon EC2 SSH key from Step 2. Then click **Launch Mindterm**.
5. Use the terminal window to run commands on the instance.

## Using a Third-Party Client with Amazon EC2 Key Pairs

When you create a stack, you can specify an Amazon EC2 SSH key that is associated by default with every instance in the stack.

## Add Stack

Name

Region

VPC **NEW**

Default Availability Zone

Default operating system

Default root device type  Instance store  EBS backed

IAM role

Default SSH key   
*Existing keys*  
*somekey*  
**None**  
Do not use a default SSH key  
Layer Dependent

Default IAM instance profile

Host name theme

Stack color 

[Advanced \*\*NEW\*\* »](#)

The **Default SSH key** list shows your AWS account's Amazon EC2keys. You can do one of the following:

- Select the appropriate key from the list.
- Select **Do not use a default SSH key** to specify no key.

If you selected **Do not use a default SSH key**, or you want to override a stack's default key, you can specify a key when you create an instance.

**PHP App Server**

No instances. [Add an instance](#).

New Existing

Host name

Size

Availability Zone

Scaling type  24/7  
 time-based  
 load-based

SSH key   
*Existing Keys*  
*somekey*  
**None**  
Do not use a default SSH key  
64bit  
32bit

Operating system

Architecture

### To use an Amazon EC2 SSH key

1. Obtain the instance's public DNS name from its details page.
2. Provide the associated private key to your preferred SSH client.

3. Enter one of the following host names, where *DNSName* is the instance's DNS name:
  - For Amazon Linux: ec2-user@*DNSName*
  - For Ubuntu: ubuntu@*DNSName*

# Apps

---

Some of AWS OpsWorks's layers support application servers. In this case, *application* includes static HTML pages.

- The Rails App Server layer serves Ruby on Rails applications.
- The PHP App Server layer serves PHP applications.
- The Node.js App Server layer serves JavaScript applications.
- The Static Web Server layer serves static HTML pages, which can include client-side scripting.
- The Java App Server layer serves JSP pages and WAR files.

An AWS OpsWorks *app* represents code that you want to run on an application server. The code itself resides in a repository such as an Amazon S3 archive ; the app contains the information required to deploy the code to the appropriate application server instances.

**Note**

If your stack includes multiple application server instances, you typically add a load balancing layer to evenly distribute the incoming traffic across the available instances. For more information, see [HAProxy AWS OpsWorks Layer \(p. 72\)](#).

**Topics**

- [Adding Apps \(p. 125\)](#)
- [Deploying Apps \(p. 132\)](#)
- [Editing Apps \(p. 134\)](#)
- [Connecting an Application to a Database Server \(p. 135\)](#)
- [Using Environment Variables \(p. 141\)](#)
- [Passing Data to Applications \(p. 143\)](#)
- [Using Repository SSH Keys \(p. 146\)](#)
- [Using Custom Domains \(p. 146\)](#)
- [Using SSL \(p. 148\)](#)

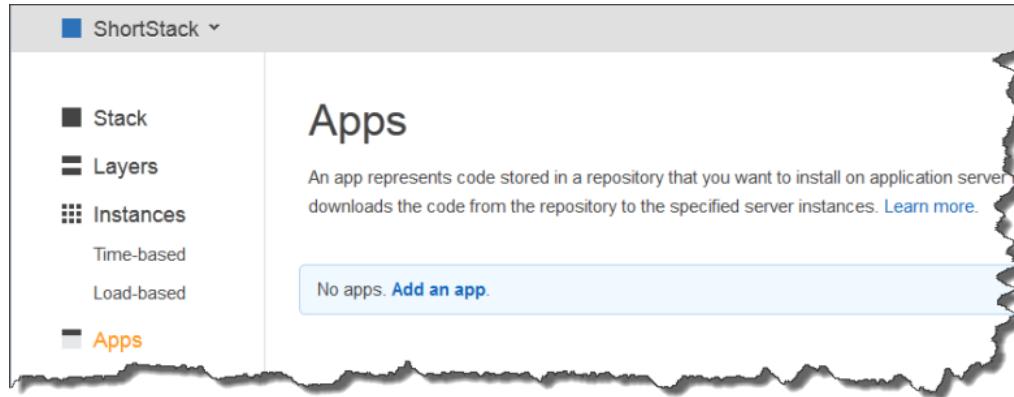
## Adding Apps

The first step in installing an application on an application server layer is to add an app to the stack. The app represents the application, and contains the information required to deploy the application to the

servers and a variety of metadata such as the application's name and type. You must have Manage permissions to add an app to a stack. For more information, see [Managing User Permissions \(p. 277\)](#).

### To add an app to a stack

1. Put the code in your preferred repository—an Amazon S3 archive, a Git repository, a Subversion repository, or an HTTP archive. For more information, see [Application Source \(p. 129\)](#).
2. Click **Apps** in the navigation pane. On the **Apps** page, click **Add an app** for your first app. For subsequent apps, click **+App**.



3. Use the **App New** page to configure the app, as described in the following section.

## Configuring an App

The **Add App** page consists of the following sections: **Settings**, **Application source**, **Data Sources**, **Environment Variables**, **Add Domains**, and **SSL Settings**.

## Add App

Settings

Name

Type

Document root

Application Source

Repository type

Repository URL

Repository SSH key

Branch/Revision

Data Sources

Data source type  RDS  OpsWorks  
 None

Environment Variables

KEY  value  Protected value

Add Domains

Domain name  Optional

SSL Settings

Enable SSL  No

[Cancel](#) [Add App](#)

### Topics

- [Settings \(p. 127\)](#)
- [Application Source \(p. 129\)](#)
- [Data Sources \(p. 130\)](#)
- [Environment Variables \(p. 131\)](#)
- [Domain and SSL Settings \(p. 132\)](#)

## Settings

Your choice for **App type** determines what other options you can set in this section.

### Topics

- [All App Types \(p. 127\)](#)
- [Ruby on Rails, PHP, Static and Other App Types \(p. 128\)](#)
- [Ruby on Rails App Type \(p. 128\)](#)

## All App Types

All app types have the following required fields:

### Name

The app name, which is used to represent the app in the UI. AWS OpsWorks also uses this name to generate a short name for the app that is used internally and to identify the app in the [stack configuration](#).

figuration and deployment JSON (p. 262). For example, the short name for an app named SimplePHPApp is simplephpapp. After you have added the app to the stack, you can see the short name by clicking Apps in the navigation pane and then clicking the app's name to open the details page.

#### App type

There are five standard types—**Ruby on Rails**, **PHP**, **Node.js** (JavaScript), **Static** (HTML), **Java**—which correspond to the standard application server layers. When you deploy the app, AWS OpsWorks installs the code and related files on the layer's instances. The **Other** app type identifies apps intended for custom application server layers.

#### Important

The built-in deployment recipes deploy only the standard app types. If you set **App Type** to **Other**, AWS OpsWorks does not deploy the app for you. You must implement your own deployment recipes and assign them to the layer's Deploy lifecycle event. For more information, see [Cookbooks and Recipes \(p. 153\)](#). For an example of how to implement Deploy recipes, see [Deploy Recipes \(p. 254\)](#).

## Ruby on Rails, PHP, Static and Other App Types

The **Ruby on Rails**, **PHP**, **Static**, and **Other** app types include an optional **Document root** setting for specifying the repository subdirectory that contains the app's code. For all app types, AWS OpsWorks assigns the **Document root** value to the [\[:document\\_root\] \(p. 396\)](#) attribute in the app's [deploy JSON \(p. 265\)](#).

#### Ruby on Rails, PHP, and Static Apps

AWS OpsWorks deploys the code from the specified repository location to the server, so that the application is served from the standard root folder. Note the following:

- If you accept the default **Document root** setting, AWS OpsWorks serves the code from the repository's parent directory.
- If you set **Document root** to a subdirectory name, AWS OpsWorks ignores the files in the repository's parent directory and serves the code from the specified subdirectory.

For example, the version of SimplePHPApp used in [Step 3: Add a Back-end Data Store \(p. 20\)](#) is in the repository's `web` subdirectory, so **Document root** is set to `web`.

#### Note

For **Ruby on Rails** apps, the **Document root** setting applies only to the directory that contains static files and assets, which is public by default.

#### Other Apps

AWS OpsWorks simply assigns the **Document root** setting to the [\[:document\\_root\] \(p. 396\)](#) attribute in the app's [deploy JSON](#). The default value is `null`. Your deployment recipes can obtain that value from the [deploy JSON](#) using standard Chef node syntax and deploy the specified code to the appropriate location on the server. For more information about how to deploy apps, see [Deploy Recipes \(p. 254\)](#).

## Ruby on Rails App Type

The Ruby on Rails app type has two additional optional settings:

- **Rails environment** specifies configuration settings for different environments, such as development, test, and production. The settings for each of the app's supported environments are specified in a file named `environment_name.rb`. To specify an environment, set **Rails environment** to the appropriate `environment_name` value (omit the `.rb` extension). The default value is `production`. For an example of environment-specific configurations, see <https://github.com/awslabs/opsworks-demo-rails-photo-share-app/tree/master/config/environments>.
- **Enable auto bundle** lets you specify whether to run `bundle install` when deploying apps. The default value is `Yes`.

## Application Source

AWS OpsWorks can deploy apps from any of five repository types: Git, Subversion, Amazon S3 bundle, HTTP bundle, and Other. All repository types require you to specify the repository type and the repository URL. Individual repository types have their own requirements, as explained below.

### Topics

- [HTTP Archive \(p. 129\)](#)
- [Amazon S3 Archive \(p. 129\)](#)
- [Git Repository \(p. 129\)](#)
- [Subversion Repository \(p. 130\)](#)
- [Other Repositories \(p. 130\)](#)

### HTTP Archive

To use a publicly-accessible HTTP server as a repository:

1. Create a compressed archive—zip, gzip, bzip2, or tarball—of the folder that contains the app's code and any associated files.
2. Upload the archive file to the server.
3. To specify the repository in the console, select HTTP Archive as the repository type and enter the URL.

If the archive is password-protected, under **Application Source**, specify the user name and password.

### Amazon S3 Archive

To use an Amazon Simple Storage Service bucket as a repository:

1. Create a public or private Amazon S3 bucket. For more information, see [Amazon S3 Documentation](#).
2. For private buckets, create an IAM user with at least read-only rights to the Amazon S3 bucket and record the access and secret keys. For more information, see [AWS Identity and Access Management \(IAM\) Documentation](#).
3. Put the code and any associated files in a folder and store the folder in a compressed archive—zip, gzip, bzip2, or tarball.
4. Upload the archive file to the Amazon S3 bucket.
5. To specify the repository in the AWS OpsWorks console:
  - For public buckets, enter the URL and specify HTTP Archive as the repository type.
  - For private buckets, enter the URL and specify S3 Archive as the repository type. Under **Application Source**, specify an **Access key ID** and **Secret access key**, such as IAM user keys, that grant you permission to access the bucket.

### Git Repository

A [Git](#) repository provides source control and versioning. AWS OpsWorks supports publicly hosted repository sites such as [GitHub](#) or [Bitbucket](#) as well as privately hosted Git servers. For both apps and Git submodules, the format you use to specify the repository's URL in **Application Source** depends on whether the repository is public or private:

**Public repository** – Use the HTTPS or Git read-only protocols. For example, [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#) uses a public GitHub repository that can be accessed by either of the following URL formats:

- Git read-only: `git://github.com/amazonwebservices/opsworks-demo-php-simple-app.git`
- HTTPS: `https://github.com/amazonwebservices/opsworks-demo-php-simple-app.git`

**Private repository** – Use the SSH read/write format shown in these examples:

- Github repositories: `git@github.com:project/repository`.
- Repositories on a Git server: `user@server:project/repository`

Selecting **Git** under **Source Control** displays two additional optional settings:

#### Repository SSH key

You must specify a deploy SSH key to access private Git repositories. For Git submodules, the specified key must have access to those submodules. For more information, see [Using Repository SSH Keys \(p. 146\)](#).

#### Important

The deploy SSH key cannot require a password; AWS OpsWorks has no way to pass it through.

#### Branch/Revision

If the repository has multiple branches, AWS OpsWorks downloads the master branch by default. To specify a particular branch, enter the branch name, SHA1 hash, or tag name. To specify a particular commit, enter the full 40-hexdigit commit id.

## Subversion Repository

A **Subversion** repository provides source control and versioning. For a private repository, you must provide your user name and password. You can also specify a revision name if you have multiple revisions. Selecting **Subversion** under **Source Control** displays three additional settings:

- **User name** – Your user name, for private repositories.
- **Password** – Your password, for private repositories.
- **Branch/Revision** – [Optional] The revision name, if you have multiple revisions.

To specify a branch or tag, you must modify the repository URL, for example: `http://repository_domain_name/repos/myapp/branches/my-apps-branch` or `http://repository_domain_name/repos/calc/myapp/my-apps-tag`.

## Other Repositories

If the standard repositories do not meet your requirements, you can use other repositories, such as [Bazaar](#). However, AWS OpsWorks does not automatically deploy apps from such repositories. You must implement custom recipes to handle the deployment process and assign them to the appropriate layers' Deploy events. For an example of how to implement Deploy recipes, see [Deploy Recipes \(p. 254\)](#).

## Data Sources

This section attaches a database to the app. You have the following options:

- **OpsWorks** – Attach an instance from the stack's [MySQL layer \(p. 80\)](#).
- **RDS** – Attach one of the stack's [Amazon RDS service layers \(p. 77\)](#).

- **None** – Do not attach a database server.

If you select **OpsWorks** or **RDS**, you must specify the following.

#### Database instance

The list includes every instance in a stack's [MySQL layer \(p. 80\)](#), if the stack has such a layer, and every Amazon RDS service layer. You can also select one of the following:

(Required) Specify which database server to attach to the app. The contents of the list depend on the data source.

- **OpsWorks** – A list of the stack's MySQL instances.
- **RDS** – A list of the stack's Amazon RDS service layers.

#### Database name

(Optional) Specify a database name.

- MySQL layer – By default, AWS OpsWorks creates a database named with the app's short name that contains no data.

You can also specify a database name, and AWS OpsWorks will create the database.

- Amazon RDS layer – Enter the database name that you specified for the Amazon RDS instance.

You can get the database name from the [Amazon RDS console](#).

When you deploy an app with an attached database, AWS OpsWorks adds the database instance's connection to the app's [deploy JSON \(p. 265\)](#). The application can then use that information to [connect to the database server \(p. 135\)](#).

To detach a database server from an app, [edit the app's configuration \(p. 134\)](#) to specify a different database server, or no server.

## Environment Variables

You can specify up to ten environment variables for each app. When you deploy the app, AWS OpsWorks defines the variables on the associated application server instances for the Java App Server, Node.js App Server, PHP App Server, and Rails App Server layers. AWS OpsWorks also stores the variables as attributes in the app's [Deploy JSON \(p. 265\)](#). For custom layers, you can have your custom recipes retrieve those values by using standard Chef node syntax. For examples of how to access an app's environment variables, see [Using Environment Variables \(p. 141\)](#).

#### Key

The variable name. It can contain up to 64 upper and lower case letters, numbers, and underscores (\_), but it must start with a letter or underscore.

#### Value

The variable's value. It can contain up to 256 characters, which must all be printable.

#### Protected value

Whether the value is protected. This setting allows you to conceal sensitive information such as passwords. If you set **Protected value** for a variable, after you create the app:

- The app's details page displays only the variable name, not the value.
- If you have permission to edit the app, you can click **Update value** to specify a new value, but you cannot see or edit the old value.

#### Important

We recommend that you do not use environment variables, even protected variables, to pass AWS credentials to an instance. A more appropriate way to provide credentials to your app is to use an IAM role. For more information, see [Specifying Permissions for Apps Running on EC2 instances \(p. 296\)](#).

## Domain and SSL Settings

The final two sections are the same for all app types.

### Domain Settings

This section has an optional **Add Domains** field for specifying domains. For more information, see [Using Custom Domains \(p. 146\)](#).

### SSL Settings

This section has an **SSL Support** toggle that you can use to enable or disable SSL. If you click **Yes**, you'll need to provide SSL certificate information.

# Deploying Apps

If you add new instances to a layer, after the Setup recipes complete, AWS OpsWorks automatically runs the Deploy recipes to deploy the appropriate apps. However, when you add, stop, or modify an app, you must deploy it manually to the online instances. You must have Manage or Deploy permissions to deploy an app. For more information, see [Managing User Permissions \(p. 277\)](#).

### Note

The built-in Deploy recipes handle only the standard app types: Rails, Node.js, PHP, Java, and static web. To use the following procedure with other app types, you must implement custom Deploy recipes and assign them to the appropriate layer's Deploy lifecycle event. For more information, see [Cookbooks and Recipes \(p. 153\)](#).

### To deploy an app

1. On the **Apps** page, click the app's **deploy** action.

## Apps

An app represents code stored in a repository that you want to install on application server instances. When you deploy the app, OpsWorks downloads the code from the repository to the specified server instances. [Learn more](#).

Name	Type	Last deployment	Actions
SimplePHP	PHP		<a href="#">deploy</a> <a href="#">edit</a> <a href="#">delete</a>
<a href="#">+ App</a>			

### Tip

You can also deploy an app by clicking **Deployments** in the navigation pane. On the **Deployments & Commands** page, click **Deploy an app**. When you do this, you can also choose which app to deploy.

2. Specify the following:

- (Required) Set **Command**: to **deploy**, if it is not already selected.
- (Optional) Include a comment.
- (Optional) For Rails apps, you can specify whether to migrate the database.
- The default setting is **No**, which does not migrate the database. Specify **Yes** to migrate the database. However, if you deploy the app to more than one Rails App Server instance, only one of the instances performs the migration.

## Deploy app

**Settings**

App	SimplePHP
Command	deploy
Comment	Optional

[Advanced >>](#)

**Instances** ⓘ

OpsWorks will run this command on **4** of **4** instances. The assigned recipes are run on all selected instances.

<input checked="" type="checkbox"/> HAProxy	<input checked="" type="checkbox"/> lb1 (online)
Click to select all instances in this layer	
<input checked="" type="checkbox"/> Ganglia	<input checked="" type="checkbox"/> monitoring-master1
Click to select all instances in this layer	(online)
<input checked="" type="checkbox"/> PHP App Server	<input checked="" type="checkbox"/> php-app1 (online) <input checked="" type="checkbox"/> php-app2 (online)
Click to select all instances in this layer	

[Cancel](#) Deploy

3. Click **Advanced >>** to specify a custom JSON object. AWS OpsWorks passes each instance a stack configuration JSON that contains the deployment details and can be used by the deploy recipes to handle installation and configuration. You can use the custom JSON field to [override default AWS OpsWorks settings \(p. 233\)](#) or pass custom settings to your custom recipes. For more information about how to use custom JSON, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#)
4. Under **Instances**, click **Advanced >>** and specify which instances to run the deploy command on.

The deploy command triggers a Deploy event, which runs the deploy recipes on the selected instances. The deploy recipes for the associated application server download the code and related files from the repository and install them on the instance, so you typically select all of the associated application server instances. However, other instance types might require some configuration changes to accommodate the new app, so it is often useful to run deploy recipes on those instances as well. Those recipes update the configuration as needed but do not install the app's files. For more information about recipes, see [Cookbooks and Recipes \(p. 153\)](#).

5. Click **Deploy** to run the deploy recipes on the specified instances, which displays the Deployment page. When the process is complete, AWS OpsWorks marks the app with a green check to indicate successful deployment. If deployment fails, AWS OpsWorks marks the app with a red X. In that case, you can go to the **Deployments** page and examine the deployment log for more information.

## Deployment SimplePHPApp - deploy

[Repeat](#)

Status	successful	User	guysm
Created at	2014-03-05 13:32:22 UTC		
Completed at	2014-03-05 13:33:04 UTC		
Duration	00:00:42		
<span style="border: 1px solid #ccc; padding: 2px 5px;">Hostname</span> <span style="border: 1px solid #ccc; padding: 2px 5px;">SSH</span> <span style="border: 1px solid #ccc; padding: 2px 5px;">Layers</span> <span style="border: 1px solid #ccc; padding: 2px 5px;">Duration</span> <span style="border: 1px solid #ccc; padding: 2px 5px;">Log</span>			
<input checked="" type="checkbox"/> php-app1	PHP App Server	00:00:42	<a href="#">show</a>

## Other Deployment Commands

The **Deploy app** page includes several other commands for managing your apps and the associated servers:

### Undeploy

Triggers an Undeploy lifecycle event (p. 176), which runs the undeploy recipes to remove all versions of the app from the specified instances.

### Rollback

Restores the previously deployed app version. For example, if you have deployed the app three times and then run **Rollback**, the server will serve the app from the second deployment. If you run **Rollback** again, the server will serve the app from the first deployment. By default, AWS OpsWorks stores the five most recent deployments, which allows you to roll back up to four versions. If you exceed the number of stored versions, the command fails and leaves the oldest version in place. You can modify the default number of stored versions by overriding one of the following AWS OpsWorks attributes.

- Override `[:opsworks][:deploy_keep_releases]` (p. 406) to change the setting for all apps.
- Override `[:deploy]['appshortname'][:keep_releases]` (p. 396) to change the setting for a particular app.

For more information on how to override AWS OpsWorks attributes, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

### Start Web Server

Runs recipes that start the application server on the specified instances.

### Stop Web Server

Runs recipes that stop the application server on the specified instances.

### Restart Web Server

Runs recipes that restart the application server on the specified instances.

### Important

**Start Web Server**, **Stop Web Server**, **Restart Web Server**, and **Rollback** are essentially customized versions of the [Execute Recipes stack command \(p. 52\)](#). They run a set of recipes that perform the task on the specified instances.

- These commands do not trigger a lifecycle event, so you cannot hook them to run custom code.
- These commands work only for the built-in [application server layers \(p. 81\)](#).

They have no effect on custom layers. To start, stop, or restart servers on a custom layer, you must install custom recipes to perform these tasks and use Execute Recipes to run them. For more information on how to implement and install recipes, see [Cookbooks and Recipes \(p. 153\)](#).

## Editing Apps

You can modify an app's configuration by editing the app. For example, if you have a new version, you can edit the app's AWS OpsWorks settings to use the new repository branch. You must have Manage or Deploy permissions to edit an app's configuration. For more information, see [Managing User Permissions \(p. 277\)](#).

### To edit an app

1. On the **Apps** page click the app name to open its details page.
2. Click **Edit** to change the app's configuration.

- If you modify the app's name, AWS OpsWorks uses the new name to identify the app in the console. Changing the name does not change the associated short name. The short name is set when you add the app to the stack and cannot be subsequently modified.
- If you have specified a protected environment variable, you cannot see or edit the value. However, you can specify a new value by clicking **Update value**.

3. Click **Save** to save the new configuration and then **Deploy App** to deploy the app.

After you have modified the app settings, you must deploy the app. When you first deploy an app, the Deploy recipes download the code and related files to the app server instances, which then run the local copy. If you modify the app in the repository, you must ensure that the updated code and related files are installed on your app server instances. AWS OpsWorks automatically deploys the current app version to new instances when they are started. For existing instances, however, the situation is different:

- You must manually deploy the updated app to online instances.
- You do not have to deploy the updated app to offline instances, including load-based and time-based instances; AWS OpsWorks automatically deploys the latest app version when they are restarted.

For more information on how to deploy apps, see [Deploying Apps \(p. 132\)](#)

## Connecting an Application to a Database Server

You associate a database server with an app when you [create the app \(p. 125\)](#) or later by [editing the app \(p. 134\)](#). Your application then uses the database connection information—user name, password, ...—to connect to the database server. AWS OpsWorks provides this information to applications in two ways:

- For the Java App Server, PHP App Server, Node.js App Server, and Rails App Server layers, AWS OpsWorks creates a file on each instance containing the connection data that the application can use to connect to the database server.

The details depend on the language. For example, for PHP applications, AWS OpsWorks puts the data in a `.php` file that you can simply include in your PHP application.

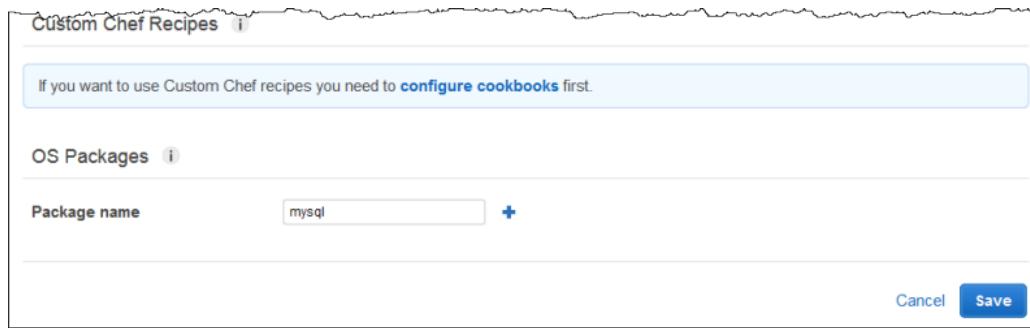
- When you deploy an app, AWS OpsWorks includes the connection information in the [stack configuration and deployment JSON \(p. 262\)](#) that is installed on each instance.

If you are using a custom layer or the language-specific files don't suit your requirements, you can implement a custom recipe to extract the connection information from the JSON and put it in an appropriate location in your preferred format.

### Important

If you want to associate an Amazon RDS service layer with your app, you must add the appropriate driver package to the associated app server layer, as follows:

1. Click **Layers** in the navigation pane and open the app server's **Recipes** tab.
2. Click **Edit** and add the appropriate driver package to **OS Packages**. For example, you should specify `mysql` if the layer contains Amazon Linux instances and `mysql-client` if the layer contains Ubuntu instances.
3. Save the changes and redeploy the app.



The following topics describe the various language-specific data files and how to use a custom recipe to extract the connection information from the stack configuration and deployment JSON.

**Note**

You can use the language-specific data files to connect to database servers on MySQL instances and to Amazon RDS instances running a MySQL database engine. You must use a custom recipe for Amazon RDS instances running a different database engine, such as Postgres.

**Topics**

- [Java App Server \(p. 136\)](#)
- [Node.js App Server \(p. 137\)](#)
- [PHP App Server \(p. 137\)](#)
- [Rails App Server \(p. 138\)](#)
- [Using a Custom Recipe \(p. 139\)](#)

## Java App Server

For Java App Server instances, the built-in Configure recipe creates a web context file named `appname.xml` and places it in Tomcat's `localhost` folder, which is typically the `/etc/tomcat7/Catalina/localhost` folder. To see the recipe (`context.rb`) and template (`webapp_context.xml.erb`), go to the [AWS OpsWorks cookbook repository](#).

For the Configure recipe to create the context file, you must [add the following custom JSON to your stack configuration JSON \(p. 53\)](#). It assigns a JDBC resource name (`jdbc/mydb`) to the web app context name (the app's short name, for this example).

```
{  
  "opsworks_java": {  
    "datasources": {  
      "app_shortname": "jdbc/mydb"  
    }  
  }  
}
```

The following is an example of a context file. For more information on context files, see [tomcat::context \(p. 251\)](#).

```
<Context>  
<Resource name="jdbc/mydb" auth="Container" type="javax.sql.DataSource"  
maxActive="20" maxIdle="5" maxWait="10000"
```

```
username="opsuser" password="AsdFGh3k"
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://opsinstance.ccdvt3hwogla.us-east-1.rds.amazonaws.com:3306/ops_db"
factory="org.apache.commons.dbcp.BasicDataSourceFactory" />
</Context>
```

Tomcat uses the data in the context file to create and initialize a `DataSource` object and bind it to a logical name. It then registers the logical name with a Java Naming and Directory Interface (JNDI) naming service. For more information, see [The SimpleJSP Application \(p. 256\)](#).

## Node.js App Server

For Node.js App Server instances, the built-in Configure recipe creates a file named `opsworks.js` and places it in the application's `shared/config` folder, which is typically `/srv/www/app_short-name/shared/config`. To see the recipe (`configure.rb`) and template (`opsworks.js.erb`), go to the [AWS OpsWorks cookbook repository](#).

The following is an example of an `opsworks.js` file.

```
exports.db = { "adapter": "mysql",
  "database": "ops_db",
  "password": "AsdFGh3k",
  "port": 3306,
  "reconnect": true,
  "username": "opsuser",
  "data_source_provider": "rds",
  "host": "opsinstance.ccdvt3hwogla.us-east-1.rds.amazonaws.com"
}

exports.memcached = { "port": 11211,
  "host": null}
```

To use this file, simply include it in your application and use the values from `exports.db` to set up the database connection.

## PHP App Server

For PHP App Server instances, the built-in Configure recipe creates a file named `opsworks.php` and places it in the application's `shared/config` folder, which is typically `/srv/www/app_short-name/shared/config`. To see the recipe (`configure.rb`) and template (`opsworks.php.erb`), go to the [AWS OpsWorks cookbook repository](#).

The following is an example of an `opsworks.php` file for an app named `SimplePHPApp`, which is located in the `/srv/www/simplephpapp/shared/config` directory.

```
<?php
class OpsWorksDb {
  public $adapter, $database, $encoding, $host, $username, $password, $reconnect;

  public function __construct() {
```

```

        $this->adapter = 'mysql';
        $this->database = 'simplephpapp';
        $this->encoding = 'utf8';
        $this->host = 'opsinstance.ccdvt3hwogla.us-east-1.rds.amazonaws.com';
        $this->username = 'opsuser';
        $this->password = 'AsdFGh3k';
        $this->reconnect = 'true';
    }
}

class OpsWorksMemcached {
    public $host, $port;

    public function __construct() {
        $this->host = '';
        $this->port = '11211';
    }
}

class OpsWorks {
    public $db;
    public $memcached;
    private $stack_map;

    public function __construct() {
        $this->db = new OpsWorksDb();
        $this->memcached = new OpsWorksMemcached();
        $this->stack_map = array('php-app' => array('10.84.78.71'), 'db-master' =>
array());
        $this->stack_name = 'RDSStack';
    }

    public function layers() {
        return array_keys($this->stack_map);
    }

    public function hosts($layer) {
        return $this->stack_map[$layer];
    }
}
?>
```

To use this file, simply include it in your application and use the values from `opsworks.php` to set up the database connection.

## Rails App Server

For Rails App Server instances, the built-in Configure recipe puts the connection information in a YAML file named `database.yml` in the app's `shared/config` folder, which is typically something like `/srv/www/app_shortname/shared/config`. To see the recipe (`configure.rb`) and template (`database.yml.erb`), go to the [AWS OpsWorks cookbook repository](#).

The following is an example of a `database.yml` file.

development:

```
adapter: "mysql"
database: "ops_db"
encoding: "utf8"
host: "opsinstance.ccdv3hwogla.us-east-1.rds.amazonaws.com"
username: "opsuser"
password: "AsdFGh3k"
reconnect: true
port: 3306
```

## Using a Custom Recipe

If you are using a custom layer, or the language-specific files don't suit your requirements, you can implement a custom recipe that extracts the connection data from the app's [deploy JSON \(p. 265\)](#) and saves it in a form that the application can read, such as a YAML file.

You attach a database server to an app when you [create the app \(p. 125\)](#) or later by [editing the app \(p. 134\)](#). When you deploy the app, AWS OpsWorks installs a [stack configuration and deployment JSON \(p. 262\)](#) on each instance. The JSON's `deploy` namespace includes an attribute for each deployed app, named with the app's short name. When you attach a database server to an app, AWS OpsWorks populates the app's `[:database]` attribute with the connection information, and installs it on the stack's instances for each subsequent deployment. The attribute values are either user-provided or generated by AWS OpsWorks.

### Note

AWS OpsWorks allows you to attach a database server to multiple apps, but each app can have only one attached database server. If you want to connect an application to more than one database server, attach one of the servers to the app, and use the information in the app's `deploy` JSON to connect to that server. Use custom JSON to pass the connection information for the other database servers to the application. For more information, see [Passing Data to Applications \(p. 143\)](#).

The following example is an excerpt of a stack configuration and deployment JSON that shows the `[:database]` attribute for the MySQL database that was used in [Step 3: Add a Back-end Data Store \(p. 20\)](#).

```
{
  ...
  "deploy": {
    "simplephpapp": {
      ...
      "database": {
        "reconnect": true,
        "database": "simplephpapp",
        "host": "10.214.175.110",
        "username": "root",
        "password": "vjud1hw5v8"
      },
      ...
    }
  }
}
```

An application can use the connection information from the instance's `deploy` JSON to connect to a database. However, applications cannot access that information directly—only recipes can access the

deploy JSON. You can address this issue by implementing a custom recipe that extracts the connection information from the deploy JSON and puts it in a file that can be read by the application. The details depend on the application type. As an example, the remainder of this topic describes how you can use a custom recipe to set up a database connection for the PHP application used in Step 2 of [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#).

To set up the connection between the PHP application server and a MySQL data store, you create a custom recipe named `appsetup.rb` and assign it to the PHP App Server layer's Deploy [lifecycle event \(p. 176\)](#). When you deploy `SimplePHPApp`, AWS OpsWorks runs `appsetup.rb`, which extracts the connection information from the deploy JSON and creates a file named `db-connect.php` in the application's current directory. The following is a slightly simplified version of the recipe. For more information, see .

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

    variables(
      :host => (deploy[:database][:host] rescue nil),
      :user => (deploy[:database][:username] rescue nil),
      :password => (deploy[:database][:password] rescue nil),
      :db => (deploy[:database][:database] rescue nil),
    )
    ...
  end
end
```

The stack has two layers, PHP App Server and MySQL, each with one instance. You create a PHP app named `SimplePHPApp` and attach a MySQL instance to the app to serve as a back-end data store.

The variables that characterize the connection—`host`, `user`, and so on—are set the corresponding values from the [deploy JSON's \(p. 265\)](#) `[:deploy][:app_name][:database]` attributes. The recipe uses these variables to construct `db-connect.php` based on the `db-connect.php.erb` template. This template simply defines a set of variables—`DB_NAME`, `DB_USER`, and so on—and sets them to the corresponding values from the deploy JSON.

```
<?php

define('DB_NAME', '<%= @db%>');
define('DB_USER', '<%= @user%>');
define('DB_PASSWORD', '<%= @password%>');
define('DB_HOST', '<%= @host%>');

?>
```

For general information about recipes and templates, see [A Short Digression: Cookbooks, Recipes, and AWS OpsWorks Attributes \(p. 23\)](#). The complete cookbook is stored at [phpapp](#).

SimplePHPApp accesses the data by requiring `db-connect.php` and using `DB_NAME` and so on to set up the connection, as shown in the following excerpt:

```
<?php

require __DIR__ . '/../vendor/autoload.php';
require __DIR__ . '/../db-connect.php';

...

// Setup the database
$app['db.dsn'] = 'mysql:dbname=' . DB_NAME . ';host=' . DB_HOST;
$app['db'] = $app->share(function ($app) {
    return new PDO($app['db.dsn'], DB_USER, DB_PASSWORD);
});
...
```

The complete application is stored at [opsworks-demo-php-simple-app](#).

**Tip**

This general approach can be used to connect an app to a variety of database servers, not just those that are managed by AWS OpsWorks, or pass other sorts of information to applications. For more information, see [Passing Data to Applications \(p. 143\)](#).

## Using Environment Variables

When you [specify environment variables for an app \(p. 131\)](#), AWS OpsWorks does the following:

- Installs the variables on built-in application server layers' instances in a form that the application server can access.

The following sections describe how each app type can obtain a variable's value.

- Adds the variable definitions to the app's [deploy JSON \(p. 265\)](#), which are then added to the node object.

Custom layers can use a recipe to retrieve a variable's value by using standard node syntax, and store it in a form that is accessible to the layer's apps.

This topic describes how different server types can access an app's environment variables.

### Topics

- [Java App Server \(p. 142\)](#)
- [Node.js App Server \(p. 142\)](#)
- [PHP App Server \(p. 142\)](#)
- [Rails App Server \(p. 142\)](#)
- [Custom App Servers \(p. 143\)](#)

## Java App Server

With Java, you can use JNDI context lookup to obtain an environment variable's value by calling the `Context.lookup` method. The following Java Server Pages (JSP) example shows how to obtain the value of the `TEST_ENV` variable.

```
<%@ page import="java.net.InetAddress, java.lang.*, javax.naming.Context,
javax.naming.InitialContext" %>
<%
Object object = ((Context)(new InitialContext().lookup("java:comp/env"))).look
up("TEST_ENV");
out.println("{\"environment\" : \"\" + object.toString().replace("\\", "\\\\")" +
"\")");
%>
```

## Node.js App Server

With Node.js, you can obtain an environment variable's value by calling `process.env`. The following example shows how to obtain the value of the `TEST_ENV` variable.

```
var express = require('express');
var app = express();
app.get('/', function(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.send(JSON.stringify({'environment': process.env.TEST_ENV})); })
app.use(express.static('public'));
app.listen(80);
console.log('Listening on port 80');
```

## PHP App Server

With PHP, you can obtain an environment variable's value by calling `getenv`. The following example shows how to obtain the value of the `TEST_ENV` variable.

```
<?php
$d = array('environment' => getenv("TEST_ENV"));
echo json_encode($d)
?>
```

## Rails App Server

With Ruby on Rails, you can obtain an environment variable's value by using `ENV[ "variable_name" ]`. The following example shows how to obtain the value of the `TEST_ENV` variable.

```
#!/usr/bin/env ruby
# encoding: UTF-8
require "sinatra"
```

```
require "json"

get "/" do
  JSON.dump(:environment => ENV['TEST_ENV'])
end
```

## Custom App Servers

With custom layers, you must implement a custom recipe that obtains the environment variable values from the instance's deploy JSON. The recipe can then store the data on the instance in a form that can be accessed by the application, such as a YAML file. An app's environment variable definitions are stored in the app's environment attribute in the stack configuration and deployment JSON, as shown in the following example.

```
{
  "ssh_users": {
  },
  "deploy": {
    "simplephpapp": {
      "application": "simplephpapp",
      "application_type": "php",
      "environment": {
        "USER_ID": "168424",
        "USER_KEY": "somepassword"
      },
      ...
    }
  }
}
```

A recipe can obtain variable values by using standard node syntax. The following example shows how to obtain the USER\_ID value from the preceding JSON and place it in the Chef log.

```
Chef::Log.info("USER_ID: #{node[:deploy]['simplephpapp'][ :environment][ :USER_ID]}")
```

For a more detailed description of how to retrieve information from the stack configuration and deployment JSON and store it on the instance, see [Passing Data to Applications \(p. 143\)](#).

## Passing Data to Applications

It is often useful to pass data such as key-value pairs to an application on the server. You can easily pass this type of data to an instance: Just use [custom JSON \(p. 53\)](#) to add the data to the [stack configuration and deployment JSON \(p. 262\)](#). AWS OpsWorks installs that JSON on each instance for each lifecycle event and exposes the contents as Chef attributes.

### Note

This topic assumes that you understand Chef attributes and how recipes can access the stack configuration and deployment JSON. For more information, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

Note, however, that although recipes can access the custom JSON data by using Chef attributes, applications cannot. One approach to getting the custom JSON data to one or more applications is to implement

a custom recipe that extracts the data from the JSON and writes it to a file that the application can read. The example in this topic shows how to write the data to a YAML file, but you can use the same basic approach for other file types, such as a JSON file.

To pass key-value data to the stack's instances, add custom JSON like the following to the stack configuration JSON. For more information about how to add custom JSON to a stack, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#).

```
{  
    "my_app_data": {  
        "app1": {  
            "key1": "value1",  
            "key2": "value2",  
            "key3": "value3"  
        },  
        "app2": {  
            "key1": "value1",  
            "key2": "value2",  
            "key3": "value3"  
        }  
    }  
}
```

The example assumes that you have two apps whose short names are `app1` and `app2`, each of which has three data values. The accompanying recipe assumes that you use the apps' short names to identify the associated data; the other names are arbitrary. For more information on app short names, see [Settings \(p. 127\)](#).

The recipe in the following example shows how to extract the data for each app from the stack configuration and deployment JSON and put it in a `.yml` file. The recipe assumes that your custom JSON contains data for each app.

```
node[:deploy].each do |app, deploy|  
  file File.join(deploy[:deploy_to], 'shared', 'config', 'app_data.yml') do  
    content YAML.dump(node[:my_app_data][app].to_hash)  
  end  
end
```

The stack configuration and deployment JSON includes a `deploy` attribute, which is set to an embedded JSON object with an attribute for each app, named with the app's short name. Each app attribute is set to an embedded JSON object whose attributes represent a variety of information about the app. This example uses the app's deployment directory, which is represented by the `[:deploy][:app_short_name][:deploy_to]` attribute. For more information on `[:deploy]`, see [Deploy Attributes \(p. 394\)](#).

For each app in `deploy`, the recipe does the following:

1. Creates a file named `app_data.yml` in the `shared/config` subdirectory of the application's `[:deploy_to]` directory.

For more information on how AWS OpsWorks installs apps, see [Deploy Recipes \(p. 254\)](#).

2. Converts the app's custom JSON to YAML and writes the formatted data to `app_data.yml`.

## To pass data to an app

1. Add an app to the stack and note its short name. For more information, see [Adding Apps \(p. 125\)](#).
2. Add custom JSON with the app's data to the stack configuration and deployment JSON, as described earlier. For more information on how to add custom JSON to a stack, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#).
3. Create a cookbook and add a recipe to it with code based on the previous example, modified as needed for the attribute names that you used in the custom JSON. For more information on how to create cookbooks and recipes, see [Cookbooks and Recipes \(p. 153\)](#). If you already have custom cookbooks for this stack, you could also add the recipe to an existing cookbook, or even add the code to an existing Deploy recipe.
4. Install the cookbook on your stack. For more information, see [Installing Custom Cookbooks \(p. 171\)](#).
5. Assign the recipe to the app server layer's Deploy lifecycle event. AWS OpsWorks will then run the recipe on each new instance, after it has booted. For more information, see [Executing Recipes \(p. 175\)](#).
6. Deploy the app, which also installs a stack configuration and deployment JSON that now contains your data.

### Note

If the data files must be in place before the app is deployed, you can also assign the recipe to the layer's Setup lifecycle event, which occurs once, right after the instance finishes booting. However, AWS OpsWorks will not have created the deployment directories yet, so your recipe should create the required directories explicitly prior to creating the data file. The following example explicitly creates the app's /shared/config directory, and then creates a data file in that directory.

```
node[:deploy].each do |app, deploy|  
  
  directory "#{deploy[:deploy_to]}/shared/config" do  
    owner "deploy"  
    group "www-data"  
    mode 0774  
    recursive true  
    action :create  
  end  
  
  file File.join(deploy[:deploy_to], 'shared', 'config', 'app_data.yml')  
  do  
    content YAML.dump(node[:my_app_data][app].to_hash)  
  end  
end
```

To load the data, you can use something like the following [Sinatra](#) code:

```
#!/usr/bin/env ruby  
# encoding: UTF-8  
require 'sinatra'  
require 'yaml'  
  
get '/' do  
  YAML.load(File.read(File.join('..', '..', 'shared', 'config', 'app_data.yml')))  
end
```

You can update the app's data values at any time by updating the custom JSON, as follows.

### To update the app data

1. Edit the custom JSON to update the data values.
2. Deploy the app again, which directs AWS OpsWorks to run the Deploy recipes on the stack's instances. The recipes will use attributes from the updated stack configuration and deployment JSON, so your custom recipe will update the data files with the current values.

## Using Repository SSH Keys

A Git repository SSH key, sometimes called a deploy SSH key, is an SSH key with no password that provides access to a private Git repository. Ideally, it doesn't belong to any specific developer. Its purpose is to allow AWS OpsWorks to asynchronously deploy apps or cookbooks from a Git repository without requiring any further input from you.

The basic procedure has two steps:

1. Create a deploy key for your Git repository.

For example, [Managing deploy keys](#) describes how to create a deploy key for a GitHub repository, and [Deployment Keys on Bitbucket](#) describes how to create a deploy key for a Bitbucket repository.

2. Enter the private key in the **Repository SSH Key** box when you add an app or specify a Git cookbook repository. For more information, see [Adding Apps \(p. 125\)](#).

The repository SSH key is passed to each instance when you deploy an app, which allows the instance to connect to the repository and download the code. The key is stored in a file that is accessible only to the root user.

**Important**

AWS OpsWorks does not support SSH key passphrases.

## Using Custom Domains

If you host a domain name with a third party, you can map that domain name to an app. The basic procedure is as follows:

1. Create a subdomain with your DNS registrar and map it to your load balancer's Elastic IP address or your app server's public IP address.
2. Update your app's configuration to point to the subdomain and redeploy the app.

**Note**

Make sure you forward your unqualified domain name (such as myapp1.example.com) to your qualified domain name (such as www.myapp1.example.com) so that both map to your app.

When you configure a domain for an app, it is listed as a server alias in the server's configuration file. If you are using a load balancer, the load balancer checks the domain name in the URL as requests come in and redirects the traffic based on the domain.

### To map a subdomain to an IP address

1. If you are using a load balancer, on the **Instances** page, click the load balancer instance to open its details page and get the instance's **Elastic IP** address. Otherwise, get the public IP address from the application server instance's details page.

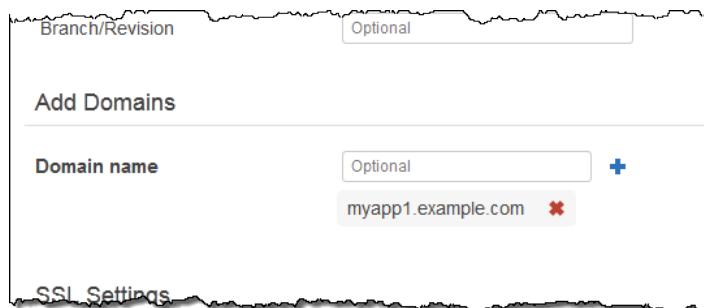
- Follow the directions provided by your DNS registrar to create and map your subdomain to the IP address from Step 1.

**Note**

If the load balancer instance terminates at some point, you are assigned a new Elastic IP address. You need to update your DNS registrar settings to map to the new Elastic IP address.

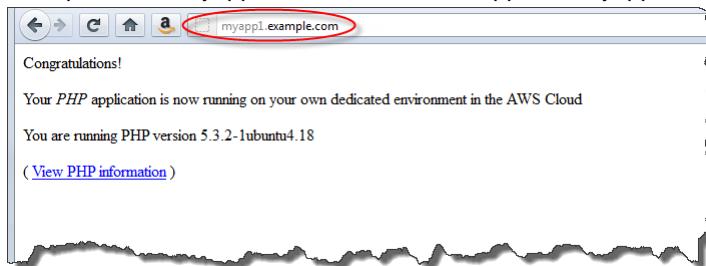
**To configure the domain for your app**

- Add the app to the stack, if you have not already done so. For more information, see [Adding Apps \(p. 125\)](#)
- On the **Apps** page click **edit** in the app's **Actions** column to edit the app's configuration.
- In the **Add Domains** section, add the domain name to the app's domains and click **+** to add the domain to the list.



- Click **Save** to save the new configuration and then **Deploy App** to deploy the app. If the app has already been deployed, this step redeploys it with the new configuration. For more information, see [Deploying Apps \(p. 132\)](#).

After your DNS settings take effect and your app has successfully been deployed or redeployed, you should be able to enter the domain in your web browser and see the application appear. The following example shows myapp1, which has been mapped to myapp1.example.com:



## Running Multiple Applications on the Same Application Server

If you have multiple applications of the same type, it is sometimes more cost-effective to run them on the same application server instances.

**To run multiple applications on the same server**

- Add an app to the stack for each application.
- Obtain a separate subdomain for each app and map the subdomains to the application server's or load balancer's IP address.

3. Edit each app's configuration to specify the appropriate subdomain.

For more information on how to perform these tasks, see [Using Custom Domains \(p. 146\)](#).

**Note**

If your application servers are running multiple HTTP applications, you can use Elastic Load Balancing or HAProxy for load-balancing. HAProxy provides more flexibility when routing requests to multiple application servers. For more information, see [HAProxy AWS OpsWorks Layer \(p. 72\)](#). For multiple HTTPS applications, you must either terminate the SSL connection at the load balancer or create a separate stack for each application. HTTPS requests are encrypted, which means that if you terminate the SSL connection at the servers, the load balancer cannot check the domain name to determine which application should handle the request.

## Using SSL

To use SSL with your application, you must obtain a digital server certificate from a Certificate Authority (CA). For the purposes of this walkthrough, we will create a certificate and then self-sign the certificate. Typically, you would only do this for testing purposes; in production, you should always get a certificate signed by a CA.

In this walkthrough, you'll do the following:

1. Install and configure OpenSSL.
2. Create a private key.
3. Create a certificate signing request.
4. Generate a self-signed certificate.
5. Edit the application with your certificate information.

**Topics**

- [Step 1: Install and Configure OpenSSL \(p. 148\)](#)
- [Step 2: Create a Private Key \(p. 149\)](#)
- [Step 3: Create a Certificate Signing Request \(p. 150\)](#)
- [Step 4: Submit the CSR to Certificate Authority \(p. 150\)](#)
- [Step 5: Edit the App \(p. 151\)](#)

## Step 1: Install and Configure OpenSSL

Creating and uploading server certificates requires a tool that supports the SSL and TLS protocols. OpenSSL is an open-source tool that provides the basic cryptographic functions necessary to create an RSA token and sign it with your private key.

The following procedure assumes that your computer does not already have OpenSSL installed.

### To install OpenSSL on Linux and Unix

1. Go to [OpenSSL: Source, Tarballs](#).
2. Download the latest source.
3. Build the package.

### To install OpenSSL on Windows

1. Go to [OpenSSL: Binary Distributions](#) and click **OpenSSL for Windows**, which displays a new page with links to the Windows downloads.
2. If not already installed on your system, select the Microsoft **Visual C++ 2008 Redistributables** link appropriate for your environment and click **Download** to save the installer locally.
3. Run the installer and follow the instructions provided by the Microsoft Visual C++ 2008 Redistributable Setup Wizard to install the redistributable.
4. Go back to [OpenSSL: Binary Distributions](#), click the appropriate version of the OpenSSL binaries for your environment, and save the installer locally.
5. Run the installer and follow the instructions in the **OpenSSL Setup Wizard** to install the binaries.

Create an environment variable that points to the OpenSSL install point by opening a terminal or command window and using the following command lines.

- On Linux and Unix

```
export OpenSSL_HOME=path_to_your_OpenSSL_installation
```

- On Windows

```
set OpenSSL_HOME=path_to_your_OpenSSL_installation
```

Add the OpenSSL binaries' path to your computer's path variable by opening a terminal or command window and using the following command lines.

- On Linux and Unix

```
export PATH=$PATH:$OpenSSL_HOME/bin
```

- On Windows

```
set Path=OpenSSL_HOME\bin;%Path%
```

#### Note

Any changes you make to the environment variables by using these command lines are valid only for the current command-line session.

## Step 2: Create a Private Key

You need a unique private key to create your Certificate Signing Request (CSR). Create the key by using the following command line:

```
openssl genrsa 2048 > privatekey.pem
```

## Step 3: Create a Certificate Signing Request

A Certificate Signing Request (CSR) is a file sent to a Certificate Authority (CA) to apply for a digital server certificate. Create the CSR by using the following command line.

```
openssl req -new -key privatekey.pem -out csr.pem
```

The command's output will look similar to the following:

```
You are about to be asked to enter information that will be incorporated  
into your certificate request.  
What you are about to enter is what is called a Distinguished Name or a DN.  
There are quite a few fields but you can leave some blank  
For some fields there will be a default value,  
If you enter '.', the field will be left blank.
```

The following table can help you create your certificate request.

### Certificate Request Data

Name	Description	Help
Country Name	The two-letter ISO abbreviation for your country. See the <a href="#">list of ISO country codes</a> .	U = et ib staS
State or Province	The name of the state or province where your organization is located. This name cannot be abbreviated.	right
Locality Name	The name of the city where your organization is located.	left
Organization Name	The full legal name of your organization. Do not abbreviate your organization name.	top
Organizational Unit	(Optional) For additional organization information.	middle
Common Name	The fully qualified domain name for your CNAME. You will receive a certificate name check warning if this is not an exact match.	middle
Email address	The server administrator's email address	mailto

#### Note

The Common Name field is often misunderstood and is completed incorrectly. The common name is typically your host plus domain name. It will look like "www.example.com" or "example.com". You need to create a CSR using your correct common name.

## Step 4: Submit the CSR to Certificate Authority

For production use, you would obtain a server certificate by submitting your CSR to a Certificate Authority (CA), which might require other credentials or proofs of identity. If your application is successful, the CA

returns digitally signed identity certificate and possibly a certificate chain file. AWS does not recommend a specific CA. For a partial listing of available CAs, see [Third-Party Certificate Authorities](#).

You can also generate a self-signed certificate, which can be used for testing purposes only. For this example, use the following command line to generate a self-signed certificate.

```
openssl x509 -req -days 365 -in csr.pem -signkey privatekey.pem -out server.crt
```

The output will look similar to the following:

```
Loading 'screen' into random state - done
Signature ok
subject=/C=us/ST=washington/L=seattle/O=corporationx/OU=marketing/CN=example.com/emailAddress=someone@example.com
Getting Private key
```

## Step 5: Edit the App

After you generate your certificate and sign it, you need to update your app to enable SSL and supply your certificate information. On the **Apps** page, click the app name to open the details page and then click **Edit App**. To enable SSL support, set **SSL Support** to **Yes**, which displays the following configuration options.

### SSL Certificate

Paste the contents of the public key certificate (.crt) file into the box. The certificate should look something like the following:

```
-----BEGIN CERTIFICATE-----
MIICuTCCAiICCQCtqFKItVQJpzANBgkqhkiG9w0BAQUFADCBoDELMAkGA1UEBhMC
dXMxEzARBgNVBAgMCndhc2hpbmD0b24xEADOBgNVBAcMB3NLYXR0bGUxDzANBgNV
BAoMBmFtYXpvbjEWMBQGA1UECwwNRGV2IGFuZCBub29sczEdMBsGA1UEAwwUc3Rl
cGhhbmllYXBpZXJjZS5jb20xIjAgBgkqhkiG9w0BCQEWE3Nhcm1lcmNlQGFtYXpv
...
-----END CERTIFICATE-----
```

#### Note

If you are using Nginx and you have a certificate chain file, you should append the contents to the public key certificate file.

If you are updating an existing certificate, do the following:

- Click **Reset now** to reset the certificate.
- If the new certificate does not match the existing private key, update **SSL Certificate Key**.
- If the new certificate does not match the existing certificate chain, update **SSL certificates of Certification Authorities**.

### SSL Certificate Key

Paste the contents of the private key file (.pem file) into the box. It should look something like the following:

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQC0CYk1JY5r4vV2NHQYEpwtsLuMMBhy1MrgBShKq+HHVLYQQCL6
+wGIiRq5qXqZ1RXje3GM5Jvcn6q0R71MfRI11FuzKyqDtneZaAIEYniZibHiUnmO
/UNqpFDosw/6hY3ONk0fSB1U4ivD0Gjpf6J80jL3DJ4R23Ed0sdL4pRT3QIDAQAB
```

```
AoGBAKmMfWrNRqYVtGKgnWB6Tji9QrKQLMXjmHeGg95mpdJELiXHhpMvrHtpIyK  
...  
-----END RSA PRIVATE KEY-----
```

### SSL certificates of Certification Authorities

If you have a certificate chain file, paste the contents into the box.

#### Note

If you are using Nginx, you should leave this box empty. If you have a certificate chain file, append it to the public key certificate file in **SSL Certificate Key**.

The screenshot shows the 'Deploy Settings' section for the 'railsapp' application. Under the 'SSL Settings' tab, the 'SSL Support' switch is set to 'On'. The 'SSL Certificate' field is empty. The 'SSL Certificate Key' field is also empty. The 'SSL Certificates of Certification Authorities' field is empty. At the bottom right, there are 'Cancel' and 'Save' buttons.

After you click **Save**, [redeploy the application \(p. 132\)](#) to update your online instances. After deployment is finished, verify that your OpenSSL installation worked, as follows.

### To verify an OpenSSL installation

1. Go to the **Instances** page.
2. Run the app by clicking the application server instance's IP address or, if you are using a load balancer, the load balancer's IP address.
3. Change the IP address prefix from `http://` to `https://` and refresh the browser to verify the page loads correctly with SSL.

# Cookbooks and Recipes

---

AWS OpsWorks uses OpsCode Chef cookbooks to handle tasks such as installing and configuring packages and deploying apps, and includes a set of built-in cookbooks that support the built-in layers. Many approaches to customizing a layer involve overriding or extending the built-in cookbooks by implementing one or more custom cookbooks. This section describes how to use custom cookbooks with AWS OpsWorks. If you are new to Chef, [Cookbooks 101 \(p. 179\)](#) provides a tutorial introduction to implementing cookbooks. For more information, see [OpsCode](#).

## Note

AWS OpsWorks currently supports Chef versions 0.9, 11.4, and 11.10. Stacks running Chef 0.9 or 11.4 use [Chef Solo](#) and stacks running Chef 11.10 use [Chef Client](#). You can use the Configuration Manager to specify which Chef version to use when you [create a stack \(p. 41\)](#). The default version is Chef 11.4. For more information, including guidelines for migrating stacks to more recent Chef versions, see [Chef Versions \(p. 161\)](#).

A cookbook typically includes the following basic components:

- **Attributes** files contain a set of attributes that represent values to be used by the recipes and templates.

For example, the built-in cookbook for the Rails App Server layer includes an attributes file with values for the Rails version, the application server stack, and so on.

- **Template** files are templates that recipes use to create other files, such as configuration files.

Template files typically let you modify the configuration file by overriding attributes—which can be done without touching the cookbook—instead of rewriting a configuration file. The standard practice is that whenever you expect to change a configuration file on an instance even slightly, you should use a template file.

- **Recipe** files are Ruby applications that define everything that is required to configure a system, including creating and configuring folders, installing and configuring packages, starting services, and so on.

Custom cookbooks don't have to have all three components. As described in [Customizing AWS OpsWorks \(p. 230\)](#), the simpler approaches to customization require only attributes or template files. In addition, cookbooks can optionally include other file types, such as definitions or specs.

## Topics

- [Cookbook Repositories \(p. 154\)](#)
- [Cookbook Components \(p. 155\)](#)
- [Chef Versions \(p. 161\)](#)
- [Ruby Versions \(p. 170\)](#)

- [Installing Custom Cookbooks \(p. 171\)](#)
- [Updating Custom Cookbooks \(p. 174\)](#)
- [Executing Recipes \(p. 175\)](#)
- [Cookbooks 101 \(p. 179\)](#)

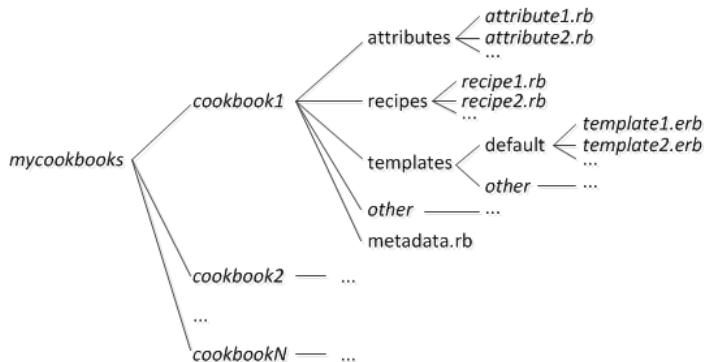
## Cookbook Repositories

Your cookbooks must be in one of four types of online repository—an Amazon S3 archive, an HTTP archive, a Git repository, or a Subversion repository. A stack can have only one custom cookbook repository, but the repository can have any number of cookbooks. When you install or update the cookbooks, AWS OpsWorks downloads the entire repository to a local cache on each of the stack's instances. When an instance needs, for example, to run one or more recipes, it uses the code from the local cache.

### Note

An archive is compressed archive of a repository, which is packaged as a zip, gzip, bzip2, or tarball file. If you use Amazon S3 or an HTTP site for your cookbook repository, AWS OpsWorks downloads the archive file to each instance and extracts the contents.

The repository must have following basic structure, where the italicized text represents user-defined directory and file names.



### Note

For HTTP and S3 archives, `mycookbooks` represents a top-level folder. For Git and Subversion repositories, `mycookbooks` represents the repository name.

Each cookbook has least one and typically all of the following standard directories, which must use the specified names:

- `attributes` – The cookbook's attributes files.
- `recipes` – The cookbook's recipe files.
- `templates` – The cookbook's template files.
- `other` – Optional user-defined directories that contain other file types, such as definitions or specs.
- `metadata.rb` – The cookbook's metadata.

For Chef 11.10 and later, if your recipes depend on other cookbooks, you must include corresponding depends statements in your cookbook's `metadata.rb` file. For example, if your cookbook includes a recipe with a statement such as `include_recipe anothercookbook::somerecipe`, your cookbook's `metadata.rb` file must include the following line: `depends "anothercookbook"`. For more information, see [About Cookbook Metadata](#).

Templates must be in a subdirectory of the `templates` directory, which contains at least one and optionally multiple subdirectories. Those subdirectories can optionally have subdirectories as well. For an example, see the [apache2](#) cookbook.

- Cookbooks usually have a `default` subdirectory, which contains the template files that Chef uses by default.
- `other` represents optional subdirectories that can be used for operating system-specific templates.
- Chef automatically uses the template from the appropriate subdirectory, based on naming conventions that are described in [Location Specificity](#). For the Amazon Linux and Ubuntu operating systems that are supported by OpsWorks, you can put operating system-specific templates in subdirectories named `amazon` or `ubuntu`, respectively. For an example, see [Tomcat Environment Configuration File \(p. 246\)](#).

**Tip**

One of the best ways to learn how to implement AWS OpsWorks cookbooks is to examine some working examples, such as the AWS OpsWorks built-in cookbooks at <https://github.com/aws/opsworks-cookbooks> or the example cookbooks at <https://github.com/amazonwebservices/opsworks-example-cookbooks>.

The details of how you handle custom cookbooks depend on your preferred repository type.

**To use an Amazon S3 or HTTP archive for your repository**

1. Implement your cookbooks by using the folder structure shown in the previous illustration.
2. Create a compressed archive—zip, gzip, bzip2, or tarball—of `mycookbooks` and upload it to the website or Amazon S3 bucket.

If you update your cookbooks, you must create and upload a new archive file.

**Note**

Your cookbooks must be in a `mycookbooks` folder, even if your repository has only one cookbook.

**To use a Github or Subversion repository**

1. Set up a repository using the structure in the previous illustration.
2. Optionally, use the repository's version control features to implement multiple branches.

If you update your cookbooks, you can do so in a new branch and just direct OpsWorks to use the new version. For details, see [Specifying a Custom Cookbook Repository \(p. 172\)](#).

[Installing Custom Cookbooks \(p. 171\)](#) describes how to direct AWS OpsWorks to install your cookbook repository on the stack's instances.

**Important**

After you update existing cookbooks, you must run the `update_cookbooks` stack command to direct AWS OpsWorks to update each instance's local cache. For more information, see [Run Stack Commands \(p. 52\)](#).

# Cookbook Components

This section describes the three standard cookbook components, attributes, templates, and recipes. For some examples of how to implement these components, see [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#). For more information, especially about how to implement recipes, see [Opscode](#).

**Topics**

- [Attributes \(p. 156\)](#)
- [Templates \(p. 158\)](#)
- [Recipes \(p. 159\)](#)

## Attributes

Recipes and templates depend on a variety of values, such as configuration settings. Rather than hardcode such values directly in recipes or templates, you can create an attributes file with an attribute that represents each value. You then use the attributes in your recipes or templates instead of explicit values. The advantage of using attributes is that you can override their values without touching the cookbook. For this reason, you should always use attributes to define the following types of values:

- Values that might vary from stack to stack or with time, such as user names.

If you hardcode such values, you must change the recipe or template each time you need to change a value. By using attributes to define these values, you can use the same cookbooks for every stack and just override the appropriate attributes.

- Sensitive values, such as passwords or secret keys.

Putting explicit sensitive values in your cookbook can increase the risk of exposure. Instead, define attributes with dummy values and override them to set the actual values. The best way to override such attributes is with custom JSON. For more information, see [Overriding Attributes Using Custom JSON \(p. 233\)](#).

For more information about attributes and how to override them, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

The following example is a portion of an attributes file (`apache.rb`) for the built-in `apache2` cookbook used by the Rails App Server layer to install and configure an Apache server.

```
...
default[:apache][:listen_ports] = [ '80','443' ]
default[:apache][:contact] = 'ops@example.com'
default[:apache][:timeout] = 120
default[:apache][:keepalive] = 'Off'
default[:apache][:keepaliverequests] = 100
default[:apache][:keepalivetimeout] = 3
default[:apache][:prefork][:startservers] = 16
default[:apache][:prefork][:minspareservers] = 16
default[:apache][:prefork][:maxspareservers] = 32
default[:apache][:prefork][:serverlimit] = 400
default[:apache][:prefork][:maxclients] = 400
default[:apache][:prefork][:maxrequestsperchild] = 10000
...
```

AWS OpsWorks defines attributes by using the following syntax:

```
node.type[:attribute][:subattribute][:...]=value
```

You can also wrap the names in quotes ("), as follows:

```
node.type[ "attribute" ][ "subattribute" ][ "..." ]=value
```

An attribute definition has the following components:

**node.**

The `node.` prefix is optional and usually omitted, as shown in the example.

**type**

The type governs whether the attribute can be overridden. AWS OpsWorks attributes typically use one of the following types:

- `default` is the most commonly used type, because it allows the attribute to be overridden.
- `set` defines an attribute that overrides one of the standard AWS OpsWorks attribute values.

**Note**

Chef supports additional types, which aren't necessary for AWS OpsWorks but might be useful for your project. For more information, see [About Attribute Files](#).

**attribute name**

The attribute name uses the standard Chef node syntax, `[:attribute][:subattribute][...]`. You can use any names you like for your attributes. However, as discussed in [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#), custom cookbook attributes are merged into the master JSON structure, along with the attributes from the built-in cookbooks and the stack configuration JSON. Commonly used configuration names such as `port` or `user` might appear in a variety of cookbooks. To avoid name collisions, the standard convention is to create qualified attribute names with at least two elements, as shown in the example. The first element should be unique and is typically based on a product name like `Apache`. It is followed by one or more subattributes that identify the particular value, such as `[:user]` or `[:port]`. You can use as many subattributes as are appropriate for your project. For example, `apache.rb` uses an `[:apache][:prefork]` prefix for the attributes that represent prefork settings.

**value**

An attribute can be set to the following types of values:

- A string, such as `default[:apache][:keepalive] = 'Off'`.
- A number (without quotes) such as `default[:apache][:timeout] = 120`.
- A Boolean value, which can be either `true` or `false` (no quotes).
- A list of values, such as `default[:apache][:listen_ports] = [ '80', '443' ]`

The attributes file is a Ruby application, so you can also use node syntax and logical operators to assign values based on other attributes. The following example is from the OpsWorks `apache2` cookbook, and assigns a value to the `[:apache][:pid_file]` attribute based on the Linux version.

```
if node['platform_version'].to_f >= 6 then
    default[:apache][:pid_file] = '/var/run/httpd/httpd.pid'
else
    default[:apache][:pid_file] = '/var/run/httpd.pid'
end
```

This particular example sets the [:apache][:pid\_file] attribute to a literal value, but you can also set attributes to other attributes. For more information about how to define attributes, see [About Attribute Files](#). For examples of working attribute files, see the AWS OpsWorks built-in cookbooks at <https://github.com/aws/opsworks-cookbooks>.

## Templates

You configure many packages by creating a configuration file and placing it in the appropriate directory. You can include a configuration file in your cookbook and copy it to the appropriate directory, but a more flexible approach is to have your recipes create the configuration file from a template. One advantage of a template is that you can use attributes to define the templates values. This allows you, for example, to modify a configuration file without touching the cookbook by using custom JSON to override the appropriate attribute values.

A template has essentially the same content and structure as the associated file. The following example is from the apache2 cookbook (apache2.conf.erb), and is used to create an Apache configuration file, httpd.conf.

```
ServerRoot "<%= node[:apache][:dir] %>
<% if node[:platform] == "debian" || node[:platform] == "ubuntu" -%>
  LockFile /var/lock/apache2/accept.lock
<% else -%>
  LockFile logs/accept.lock
<% end -%>
PidFile <%= node[:apache][:pid_file] %>
Timeout <%= node[:apache][:timeout] %>
KeepAlive <%= node[:apache][:keepalive] %>
MaxKeepAliveRequests <%= node[:apache][:keepaliverequests] %>
KeepAliveTimeout <%= node[:apache][:keepalivetimeout] %>
<IfModule mpm_prefork_module>
  StartServers      <%= node[:apache][:prefork][:startservers] %>
  MinSpareServers   <%= node[:apache][:prefork][:minspareservers] %>
  MaxSpareServers   <%= node[:apache][:prefork][:maxspareservers] %>
  ServerLimit       <%= node[:apache][:prefork][:serverlimit] %>
  MaxClients        <%= node[:apache][:prefork][:maxclients] %>
  MaxRequestsPerChild <%= node[:apache][:prefork][:maxrequestsperchild] %>
</IfModule>
...
"
```

The following example is the httpd.conf file that was generated for a Ubuntu instance:

```
ServerRoot "/etc/httpd"
LockFile logs/accept.lock
PidFile /var/run/httpd/httpd.pid
Timeout 120
KeepAlive Off
MaxKeepAliveRequests 100
KeepAliveTimeout 3
<IfModule mpm_prefork_module>
  StartServers      16
  MinSpareServers   16
  MaxSpareServers   32
  ServerLimit       400
  MaxClients        400

```

```
    MaxRequestsPerChild 10000
</IfModule>
...

```

Much of the template's text is simply copied from the template to the `httpd.conf` file. However, `<% ... %>` content is handled as follows:

- Chef replaces `<%= node[:attribute][:sub_attribute]...%>` with the attribute's value.  
For example, `StartServers <%= node[:apache][:prefork][:startservers] %>` becomes `StartServers 16` in the `httpd.conf`.
- You can use `<%if-%>`, `<%else-%>`, and `<%end-%>` to conditionally select a value.

The example sets a different file path for `accept.lock` depending on the platform.

#### Note

You are not limited to the attributes in your cookbook's attributes files. You can use any attribute in the master JSON. For example, the `node[:platform]` attribute used in the example is not an `apache2` attribute. It is one of a set of attributes that are generated by a Chef tool called `Ohai` and also incorporated into the master JSON. For more information on attributes, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

For more information on templates, including how to incorporate Ruby code, see [About Templates](#).

## Recipes

Recipes are Ruby applications that define everything that is required to configure a system. They install packages, create configuration files from templates, execute shell commands, create files and folders, and so on. You typically have AWS OpsWorks execute recipes automatically when a [lifecycle event \(p. 176\)](#) occurs on the instance but you can also run them explicitly at any time by using the `execute_recipes` stack command ([p. 175](#)). For more information, see [About Recipes](#).

A recipe typically consists largely of a series of *resources*, each of which represents an aspect of the system. Each resource includes a set of attributes that define the resource and specify what action is to be taken. Chef associates each resource with an appropriate *provider* that performs the action. For more information, see [Resources and Providers Reference](#).

A package resource helps you manage software packages. The following example is at the beginning of `default.rb` and installs the Apache package.

```
...
package 'apache2' do
  case node[:platform]
  when 'centos', 'redhat', 'fedora', 'amazon'
    package_name 'httpd'
  when 'debian', 'ubuntu'
    package_name 'apache2'
  end
  action :install
end
...
```

Chef uses the appropriate package provider for the platform. Resource attributes are often just assigned a value, but you can use Ruby logical operations to perform conditional assignments. The example uses

a case operator, which uses `node[:platform]` to identify the instance's operating system and sets the `package_name` attribute accordingly. You can insert attributes into a recipe by using the standard Chef node syntax and Chef replaces it with the associated value. You can use any attribute in the master JSON structure, not just your cookbook's attributes.

After determining the appropriate package name, the code segment ends with an `install` action, which installs the package. Other actions for this resource include `upgrade` and `remove`. For more information, see [package](#).

It is often useful to break complex installation and configuration tasks into one or more subtasks, each implemented as a separate recipe, and have your primary recipe run them at the appropriate time. The following example shows the line of code that follows the preceding example:

```
include_recipe 'apache2::service'
```

To have a recipe execute a child recipe, use the `include_recipe` keyword, followed by the recipe name. Recipes are identified by using the standard Chef `CookbookName::RecipeName` syntax, where `RecipeName` omits the `.rb` extension. The example includes the apache2 cookbook's `service.rb` recipe, which contains a `service` resource that starts the Apache server.

#### Note

An `include_recipe` statement effectively executes the recipe at that point in the primary recipe. However, what actually happens is that Chef replaces each `include_recipe` statement with the specified recipe's code before it executes the primary recipe.

A `directory` resource represents a directory, such as the one that is to contain a package's files. The following `default.rb` resource creates a log directory.

```
directory node[:apache][:log_dir] do
  mode 0755
  action :create
end
```

The log directory is defined in one of the cookbook's attributes files. The resource specifies the directory's mode as 0755, and uses a `create` action to create the directory. For more information, see [directory](#).

The `execute` resource represents commands, such as shell commands or scripts. The following example generates module.load files.

```
execute 'generate-module-list' do
  if node[:kernel][:machine] == 'x86_64'
    libdir = 'lib64'
  else
    libdir = 'lib'
  end
  command "/usr/local/bin/apache2_module_conf_generate.pl /usr/#{libdir}/ht
tpd/modules /etc/httpd/mods-available"
  action :run
end
```

The resource first determines the CPU type. `[:kernel][:machine]` is another of the automatic attributes that Chef generates to represent various system properties, the CPU type in this case. It then specifies the command, a Perl script and uses a `run` action to run the script, which generates the module.load files. For more information, see [execute](#).

A template resource represents a file—typically a configuration file—that is to be generated from one of the cookbook's template files. The following example creates an `httpd.conf` configuration file from the `apache2.conf.erb` template that was discussed in [Templates \(p. 158\)](#).

```
template 'apache2.conf' do
  case node[:platform]
  when 'centos', 'redhat', 'fedora', 'amazon'
    path "#{node[:apache][:dir]}/conf/httpd.conf"
  when 'debian', 'ubuntu'
    path "#{node[:apache][:dir]}/apache2.conf"
  end
  source 'apache2.conf.erb'
  owner 'root'
  group 'root'
  mode 0644
  notifies :restart, resources(:service => 'apache2')
end
```

The resource determines the generated file's name and location based on the instance's operating system. It then specifies `apache2.conf.erb` as the template to be used to generate the file and sets the file's owner, group, and mode. It runs the `notify` action to notify the `service` resource that represents the Apache server to restart the server. For more information, see [template](#).

## Chef Versions

AWS OpsWorks supports three versions of Chef. You select the version when you [create the stack \(p. 41\)](#). AWS OpsWorks then installs that version of Chef on all of the stacks instances along with a set of built-in recipes that are compatible with that version. If you install any custom recipes, they must be compatible with the stack's Chef version.

### Chef 0.9

AWS OpsWorks originally supported only Chef 0.9, which runs with Ruby 1.8.7. You can still specify Chef 0.9 for your stack, but new features are not available for Chef 0.9 stacks, and support is scheduled to end on July 24, 2014. We do not recommend using Chef 0.9 for new stacks, and we recommend migrating your existing Chef 0.9 stacks to Chef 11.10 as soon as possible.

### Chef 11.4

Chef 11.4 support was introduced in July 2013, and runs with Ruby 1.8.7. You can use many community cookbooks with Chef 11.4 stacks. However AWS OpsWorks uses `chef-solo` for these stacks, so there is no support for Chef search or data bags. You can often use community cookbooks that depend on those features with AWS OpsWorks, but you must modify them as described in [Implementing Recipes for Chef 11.4 Stacks \(p. 162\)](#).

### Chef 11.10

Chef 11.10 support was introduced in March 2014, and runs with Ruby 2.0.0. Chef 11.10 stacks use `chef-client in local mode`, which launches a local in-memory Chef server called `chef-zero`. The presence of this server enables recipes to use Chef search and data bags. The support has some limitations, which are described in [Implementing Recipes for Chef 11.10 Stacks \(p. 163\)](#), but you can run many community cookbooks without modification. You can also use `Berkshelf` to manage your cookbooks. AWS OpsWorks supports Berkshelf versions 2.0.14, 3.0.1, and 3.1.1.

If you want to use community cookbooks with AWS OpsWorks, we recommend that you specify [Chef 11.10 \(p. 41\)](#) for new stacks and migrate your existing stacks to Chef 11.10. For more information on how to implement recipes for Chef 11.10 stacks, see [Implementing Recipes for Chef 11.10 Stacks \(p. 163\)](#).

You can use the AWS OpsWorks console, API, or CLI to migrate your existing stacks to a newer Chef version. For more information, see [Migrating an Existing Stack to a new Chef Version \(p. 169\)](#).

### Topics

- [Implementing Recipes for Chef 11.4 Stacks \(p. 162\)](#)
- [Implementing Recipes for Chef 11.10 Stacks \(p. 163\)](#)
- [Migrating an Existing Stack to a new Chef Version \(p. 169\)](#)

## Implementing Recipes for Chef 11.4 Stacks

The primary limitation of Chef 11.4 stacks is that recipes cannot use Chef search or data bags. However, AWS OpsWorks installs [stack configuration and deployment JSON \(p. 262\)](#) on each instance that contains much of the information that you would obtain with search, including the following:

- User-defined data from the console such as host or app names.
- Stack configuration data generated by the AWS OpsWorks service, such as the stack's layers, apps, and instances, and details about each instance such as the IP address.
- Custom JSON attributes that contain data provided by the user and can serve much the same purpose as data bags.

AWS OpsWorks installs a current copy of the stack configuration and deployment JSON on each instance for each lifecycle event, prior to starting the event's Chef run. The data in the JSON is incorporated into the Chef node object, and is available to recipes through the standard `node[:attribute][:child_attribute]...` syntax.`

For example, the stack configuration and deployment JSON includes the stack name, which is represented by `node[:opsworks][:stack][:name]`. The following excerpt from one of the built-in recipes obtains the stack name and uses it to create a configuration file.

```
template '/etc/ganglia/gmetad.conf' do
  source 'gmetad.conf.erb'
  mode '0644'
  variables :stack_name => node[:opsworks][:stack][:name]
  notifies :restart, "service[gmetad]"
end
```

Many of the stack and configuration JSON attributes are assigned embedded JSON structures, which contain multiple attributes. You must iterate over these attributes to obtain the information you need. For example, `[:opsworks][:layers]` attribute contains an attribute for each layer. In the excerpt below, the stack and configuration JSON contains a top-level attribute, `deploy`.

```
{
  ...
  "deploy": {
    "app1_shortname": {
      "document_root": "app1_root",
      "deploy_to": "deploy_directory",
      "application_type": "php",
      ...
    },
    "app2_shortname": {
```

```
        "document_root": "app2_root",
        ...
    },
    ...
}
```

Each app is characterized by a set of attributes. For example, the `deploy_to` attribute represents the app's deploy directory. The following excerpt sets the user, group, and path for each app's deploy directory.

```
node[:deploy].each do |application, deploy|
  opsworks_deploy_dir do
    user deploy[:user]
    group deploy[:group]
    path deploy[:deploy_to]
  end
  ...
end
```

For more information on the stack configuration and deployment JSON, see [Customizing AWS OpsWorks \(p. 230\)](#). For more information on deploy directories, see [Deploy Recipes \(p. 254\)](#).

Chef 11.4 stacks do not support data bags, but you can add arbitrary data to the stack configuration and deployment JSON by specifying [custom JSON \(p. 53\)](#). Your recipes can then access the data by using standard Chef node syntax. For more information, see [Overriding Attributes Using Custom JSON \(p. 233\)](#).

If you need the functionality of an encrypted data bag, one option is to store sensitive attributes in a secure location such as a private Amazon S3 bucket. Your recipes can then use the [AWS Ruby SDK](#)—which is installed on all AWS OpsWorks instances—to download the data from the bucket.

**Note**

Each AWS OpsWorks instance has an instance profile. The associated [IAM role](#) specifies which AWS resources can be accessed by applications that are running on the instance. For your recipes to access an Amazon S3 bucket, the role's policy must include a statement similar to the following, which grants permission to retrieve files from a specified bucket.

```
"Action": ["s3:GetObject"],
"Effect": "Allow",
"Resource": "arn:aws:s3:::yourbucketname/*",
```

For more information on instance profiles, see [Specifying Permissions for Apps Running on EC2 instances \(p. 296\)](#).

## Implementing Recipes for Chef 11.10 Stacks

Chef 11.10 stacks provide the following advantages over Chef 11.4 stacks:

- Chef runs use Ruby 2.0.0, so your recipes can use the new Ruby syntax.
- Recipes can use Chef search and data bags.

Chef 11.10 stacks can use many community cookbooks without modification.

- You can use Berkshelf to manage cookbooks.

Berkshelf provides a much more flexible way to manage your custom cookbooks and to use community cookbooks in a stack.

- Cookbooks must declare dependencies in `metadata.rb`.

If your cookbook depends on another cookbook, you must include that dependency in your cookbook's `metadata.rb` file. For example, if your cookbook includes a recipe with a statement such as `include_recipe anothercookbook::somerecipe`, your cookbook's `metadata.rb` file must include the following line: `depends "anothercookbook"`.

- AWS OpsWorks installs a MySQL client on a stack's instances only if the stack includes a MySQL layer.
- AWS OpsWorks installs a Ganglia client on a stack's instances only if the stack includes a Ganglia layer.
- If a deployment runs `bundle install` and the install fails, the deployment also fails.

**Note**

Cookbooks with non-ASCII characters that run successfully on Chef 0.9 and 11.4 stacks might fail on a Chef 11.10 stack. The reason is that Chef 11.10 stacks use Ruby 2.0.0 for Chef runs, which is much stricter about encoding than Ruby 1.8.7. To ensure that such cookbooks run successfully on Chef 11.10 stacks, each file that uses non-ASCII characters should have a comment at the top that provides a hint about the encoding. For example, for UTF-8 encoding, the comment would be `# encoding: UTF-8`. For more information on Ruby 2.0.0 encoding, see [Encoding](#).

**Topics**

- [Cookbook Installation and Precedence \(p. 164\)](#)
- [Using Chef Search \(p. 165\)](#)
- [Using Data Bags \(p. 166\)](#)
- [Using Berkshelf \(p. 167\)](#)

## Cookbook Installation and Precedence

The procedure for installing AWS OpsWorks cookbooks works somewhat differently for Chef 11.10 stacks than for earlier Chef versions. For Chef 11.10 stacks, after AWS OpsWorks installs the built-in, custom, and Berkshelf cookbooks, it merges them to a common directory in the following order:

1. Built-in cookbooks.
2. Berkshelf cookbooks, if any.
3. Custom cookbooks, if any.

When AWS OpsWorks performs this merge, it copies the entire contents of the directories, including recipes. If there are any duplicates, the following rules apply:

- The contents of Berkshelf cookbooks take precedence over the built-in cookbooks.
- The contents of custom cookbooks take precedence over the Berkshelf cookbooks.

To illustrate how this process works, consider the following scenario, where all three cookbook directories include a cookbook named `mycookbook`:

- Built-in cookbooks – `mycookbook` includes an attributes file named `someattributes.rb`, a template file named `sometemplate.erb`, and a recipe named `somerecipe.rb`.
- Berkshelf cookbooks – `mycookbook` includes `sometemplate.erb` and `somerecipe.rb`.
- Custom cookbooks – `mycookbook` includes `somerecipe.rb`.

The merged cookbook contains the following:

- `someattributes.rb` from the built-in cookbook.
- `sometemplate.erb` from the Berkshelf cookbook.
- `somerecipe.rb` from the custom cookbook.

**Important**

You should not customize your Chef 11.10 stack by copying an entire built-in cookbook to your repository and then modifying parts of the cookbook. Doing so effectively overrides the entire built-in cookbook, including recipes. If AWS OpsWorks updates that cookbook, your stack will not get the benefit of those updates unless you manually update your private copy. For more information on how to customize stacks, see [Customizing AWS OpsWorks \(p. 230\)](#).

## Using Chef Search

You can use the Chef [search Method](#) in your recipes to query for stack data. You use the same syntax as you would for Chef server, but AWS OpsWorks obtains the data from the local node object instead of querying a Chef server. This data includes:

- The attributes from the instance's [stack configuration and deployment JSON \(p. 53\)](#).
- The attributes from the instance's built-in and custom cookbooks' attributes files.
- System data collected by Ohai.

The stack configuration and deployment JSON attributes contain most of the information that recipes typically obtain through search, including data such as host names and IP addresses for each online instance in the stack. AWS OpsWorks updates this JSON for each [lifecycle event \(p. 176\)](#), which ensures that it accurately reflects the current stack state. This means that you can often use search-dependent community recipes in your stack without modification. The search method still returns the appropriate data; it's just coming from the stack configuration and deployment JSON instead of a server.

The primary limitation of AWS OpsWorks search is that handles only the data in the local node object, the stack configuration and deployment JSON in particular. For that reason, the following types of data might not be available through search:

- Locally defined attributes on other instances.

If a recipe defines an attribute locally, that information is not reported back to the AWS OpsWorks service, so you cannot access that data from other instances by using search.

- Custom attributes in the deployment JSON.

You can specify custom JSON when you [deploy an app \(p. 132\)](#). However, if you deploy only to selected instances, the JSON is installed on only those instances. Queries for those custom JSON attributes will fail on all other instances. In addition, the custom attributes are included in the stack configuration and deployment JSON only for that particular deployment. They are accessible only until the next lifecycle event installs a new stack configuration and deployment JSON. Note that if you [specify custom JSON for the stack \(p. 53\)](#), the attributes are installed on every instance for every lifecycle event and are always accessible through search.

- Ohai data from other instances.

The Ohai tool obtains a variety of system data on an instance and makes it available to recipes. However, this data is stored locally and not reported back to the AWS OpsWorks service, so search can't access Ohai data from other instances. However, some of this data might be included in the stack configuration and deployment JSON.

- Offline instances.

The stack configuration and deployment JSON contains data only for online instances.

The following recipe excerpt shows how to get the private IP address of a PHP layer's instance by using search. Note that with Chef search, you use `role` rather than `layer` to specify a layer.

```
appserver = search(:node, "role:php-app").first
Chef::Log.info("The private IP is '#{appserver[:private_ip]}")
```

## Using Data Bags

You can use the Chef [data\\_bag\\_item method](#) in your recipes to query for information in a data bag. You use the same syntax as you would for Chef server, but AWS OpsWorks obtains the data from the instance's stack configuration and deployment JSON. However, AWS OpsWorks does not currently support Chef environments, so `node.chef_environment` always returns `_default`.

You create a data bag by using custom JSON to add one or more attributes to the stack configuration and deployment JSON `[:opsworks][:data_bags]` attribute. The following example shows the general format for creating a data bag in custom JSON.

**Note**

You cannot create a data bag by adding it to your cookbook repository. You must use custom JSON.

```
{
  "opsworks": {
    "data_bags": {
      "bag_name1": {
        "item_name1": {
          "key1" : "value1",
          "key2" : "value2",
          ...
        }
      },
      "bag_name2": {
        "item_name1": {
          "key1" : "value1",
          "key2" : "value2",
          ...
        }
      },
      ...
    }
  }
}
```

You typically [specify custom JSON for the stack \(p. 53\)](#), which installs the custom attributes on every instance for each subsequent lifecycle event. You can also specify custom JSON when you deploy an app, but those attributes are installed only for that deployment, and might be installed to only a selected set of instances. For more information, see [Deploying Apps \(p. 132\)](#).

The following custom JSON example creates data bag named `myapp`. It has one item, `mysql`, with two key-value pairs.

```
{ "opsworks": {  
    "data_bags": {  
        "myapp": {  
            "mysql": {  
                "username": "default-user",  
                "password": "default-pass"  
            }  
        }  
    }  
}
```

To use the data in your recipe, you can call `data_bag_item` and pass it the data bag and value names, as shown in the following excerpt.

```
mything = data_bag_item("myapp", "mysql")  
Chef::Log.info("The username is '#{mything['username']}'")
```

To modify the data in the data bag, just modify the custom JSON, and it will be installed on the stack's instances for the next lifecycle event.

## Using Berkshelf

With Chef 0.9 and Chef 11.4 stacks, you can install only one custom cookbook repository. With Chef 11.10 stacks, you can use [Berkshelf](#) to manage your cookbooks, which allows you to install cookbooks from multiple repositories. In particular, with Berkshelf, you can install AWS OpsWorks-compatible community cookbooks directly from their repositories instead of having to copy them to your custom cookbook repository. AWS OpsWorks supports Berkshelf versions 2.0.14, 3.0.1, and 3.1.1.

To use Berkshelf, you must explicitly enable it, as described in [Installing Custom Cookbooks \(p. 171\)](#). Then, include a `Berksfile` file in your cookbook repository's root directory that specifies which cookbooks to install. A custom cookbook repository can contain custom cookbooks in addition to a `Berksfile`. In that case, AWS OpsWorks installs both sets of cookbooks, which means that an instance can have as many as three cookbook repositories.

- The built-in cookbooks are installed to `/opt/aws/opsworks/current/cookbooks`.
- If your custom cookbook repository contains cookbooks, they are installed to `/opt/aws/opsworks/current/site-cookbooks`.
- If you have enabled Berkshelf and your custom cookbook repository contains a `Berksfile`, the specified cookbooks are installed to `/opt/aws/opsworks/current/berkshelf-cookbooks`.

To specify a cookbook source in a `Berksfile`, the simplest approach is to include a line for each cookbook in the following format:

```
cookbook 'cookbook_name', ['>= cookbook_version'], [cookbook_options]
```

The fields following `cookbook` specify the particular cookbook.

- `cookbook_name` – (Required) Specifies the cookbook's name.

If you don't include any other fields, Berkshelf installs the cookbook from the Chef community repository on GitHub, <http://cookbooks.opscode.com/api/v1/cookbooks>.

- *cookbook\_version* – (Optional) Specifies the cookbook version or versions.

You can use a prefix such as `=` or `>=` to specify a particular version or a range of acceptable versions. If you don't specify a version, Berkshelf installs the latest one.

- *cookbook\_options* – (Optional) The final field is a hash containing one or more key-value pairs that specify options such as the repository location.

For example, you can include a `git` key to designate a particular Git repository and a `tag` key to designate a particular repository branch. Specifying the repository branch is usually the best way to ensure that you install your preferred cookbook.

#### **Note**

Your repository's root directory does not have a `metadata.rb` file, so you cannot specify dependencies in the Berksfile by using `metadata`.

The following is an example of a Berksfile that shows different ways to specify cookbooks. For more information on how to create a Berksfile, see [Berkshelf](#).

```
cookbook 'apt'
cookbook 'bluepill', '>= 2.3.1'
cookbook 'ark', git: 'git://github.com/opscode-cookbooks/ark.git'
cookbook 'build-essential', '>= 1.4.2', git: 'git://github.com/opscode-cookbooks/build-essential.git', tag: 'v1.4.2'
```

This file installs the following cookbooks:

1. The most recent version of `apt` from the community cookbooks repository.
2. The most recent version `bluepill` from the community cookbooks, as long as it is version 2.3.1 or later.
3. The most recent version of `ark` from a specified repository.

The URL for this example is for a public community cookbook repository on GitHub, but you can install cookbooks from other repositories, including private repositories. For more information, see [Berkshelf](#).

4. The `build-essential` cookbook from the v1.4.2 branch of the specified repository.

The built-in cookbooks and your custom cookbooks are installed on each instance during setup and are not subsequently updated unless you manually run the [Update Custom Cookbooks stack command \(p. 52\)](#). AWS OpsWorks runs `berks install` for every Chef run, so your Berkshelf cookbooks are updated for each [lifecycle event \(p. 176\)](#), according to the following rules:

- If you have a new cookbook version in the repository, this operation updates the cookbook from the repository.
- Otherwise, this operation updates the Berkshelf cookbooks from a local cache.

#### **Note**

The operation overwrites the Berkshelf cookbooks, so if you have modified the local copies of any cookbooks, the changes will be overwritten. For more information, see [Berkshelf](#)

You can also update your Berkshelf cookbooks by running the [Update Custom Cookbooks](#) stack command, which updates both the Berkshelf cookbooks and your custom cookbooks.

# Migrating an Existing Stack to a new Chef Version

You can use AWS OpsWorks console, API, or CLI to migrate your stacks to a newer Chef version. However, your recipes might require modification to be compatible with the newer version. When preparing to migrate a stack, consider the following.

- The transition from Chef 0.9 to Chef 11.4 and from Chef 11.4 to Chef 11.10 involves a number of changes, some of them breaking changes.

For more information on the Chef 0.9 to Chef 11.4 transition, see [Implementing Recipes for Chef 11.4 Stacks \(p. 162\)](#). For more information on the Chef 11.4 to Chef 11.10 transition, see [Implementing Recipes for Chef 11.10 Stacks \(p. 163\)](#).

- Chef runs use a different Ruby version on Chef 0.9 and Chef 11.4 stacks (Ruby 1.8.7) than on Chef 11.10 stacks (Ruby 2.0.0).

For more information, see [Ruby Versions \(p. 170\)](#).

- Chef 11.10 stacks handle cookbook installation differently from Chef 0.9 or Chef 11.4 stacks.

This difference could cause problems when migrating stacks that use custom cookbooks to Chef 11.10. For more information, see [Cookbook Installation and Precedence \(p. 164\)](#).

The following are recommended guidelines for migrating a Chef stack to a newer Chef version:

## To migrate a stack to a newer Chef version

1. [Clone your production stack \(p. 51\)](#). On the **Clone Stack** page, click **Advanced>>** to display the **Configuration Management** section, and change **Chef version** to the next higher version.

### Note

If you are starting with a Chef 0.9 stack, you cannot upgrade directly to Chef 11.10; you must first upgrade to Chef 11.4. If you want to migrate your stack to Chef 11.10 before testing your recipes, wait 20 minutes for the update to be executed, and then upgrade the stack from 11.4 to 11.10.

2. Add instances to the layers and test the cloned stack's applications and cookbooks on a testing or staging system. For information about Chef 11.4 or Chef 11.10, see [All about Chef ....](#)
3. When the test results are satisfactory, do one of the following:
  - If this is your desired Chef version, you can use the cloned stack as your production stack, or reset the Chef version on your production stack.
  - If you are migrating a Chef 0.9 stack to Chef 11.10 in two stages, repeat the process to migrate the stack from Chef 11.4 to Chef 11.10.

### Tip

When you are testing recipes, you can [use SSH to connect to \(p. 119\)](#) the instance and then use the [Instance Agent CLI \(p. 368\)](#) [run\\_command \(p. 373\)](#) command to run the recipes associated with the various lifecycle events. The agent CLI is especially useful for testing Setup recipes because you can use it even if Setup fails and the instance does not reach the online state. You can also use the [Setup stack command \(p. 52\)](#) to rerun Setup recipes, but that command is only available if Setup succeeded and the instance is online.

It is possible to update a running stack to a new Chef version.

### To update a running stack to a new Chef version

1. [Edit the stack \(p. 50\)](#) to change the **Chef version** stack setting.
2. Save the new settings and wait for AWS OpsWorks to update the instances, which typically takes 15 - 20 minutes.

#### Important

AWS OpsWorks does not synchronize the Chef version update with lifecycle events. If you want to update the Chef version on a production stack, you must take care to ensure that the update is complete before the next [lifecycle event \(p. 176\)](#) occurs. If an event occurs—typically a Deploy or Configure event—the instance agent updates your custom cookbooks and runs the event's assigned recipes, whether the version update is complete or not. There is no direct way to determine when the version update is complete, but deployment logs include the Chef version.

## Ruby Versions

All instances in a stack have two Ruby versions installed.

- AWS OpsWorks installs a Ruby package on each instance, which it uses to run Chef recipes and the instance agent.

AWS OpsWorks determines the Ruby version based on which Chef version the stack is running. Do not attempt to modify this version; doing so might disable the instance agent.

- AWS OpsWorks installs a separate Ruby package on that layer's instances, to be used by your apps.

By default, AWS OpsWorks installs Ruby 2.0.0 for use by apps, or you can specify a version. The available versions depend on the stack's Chef version and are summarized later.

You can specify which Ruby version is installed for your apps as follows:

- If your stack includes a Rails App Server layer, specify the Ruby version when you [create the layer \(p. 92\)](#).
- Override the AWS OpsWorks `[:opsworks][:ruby_version]` attribute to specify your preferred Ruby version.

For example, the following custom JSON sets the app Ruby version to 1.9.3. For more information about overriding AWS OpsWorks attributes, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

```
{  
  "opsworks": {  
    "ruby_version": "1.9.3"  
  }  
}
```

The following table summarizes AWS OpsWorks Ruby versions.

Chef Version	Chef Ruby Version	Available App Ruby Versions
0.9	1.8.7	1.8.7, 1.9.3, 2.0.0
11.4	1.8.7	1.8.7, 1.9.3, 2.0.0, 2.1.x

Chef Version	Chef Ruby Version	Available App Ruby Versions
11.10	2.0.0	1.9.3, 2.0.0, 2.1.x

The install locations depend on the Chef version:

- Apps use the `/usr/local/bin/ruby` executable for all Chef Versions.
- For Chef 0.9 and 11.4, the instance agent and Chef recipes use the `/usr/bin/ruby` executable.
- For Chef 11.10, the instance agent and Chef recipes use the `/opt/aws/opsworks/local/bin/ruby` executable.

## Installing Custom Cookbooks

To have a stack install and use custom cookbooks, you must explicitly enable them.

### To enable custom cookbooks

1. On your stack's page, click **Stack Settings** to display its **Settings** page., Click **Edit** to edit the settings and scroll down to the **Configuration Management** section.
2. Toggle **Use custom Chef cookbooks** to **Yes**.

The screenshot shows the 'Configuration Management' section of the AWS OpsWorks stack settings. It includes fields for Chef version (radio buttons for 11.10, 11.4, and 0.9), Use custom Chef cookbooks (checkbox set to Yes), Repository type (dropdown set to Git), Repository URL (text input https://github.com/user/cookbooks.git), Repository SSH key (text area labeled Optional), Branch/Revision (text area labeled Optional), Manage Berkshelf (checkbox set to Yes), Berkshelf version (text input 3.1.1), Custom JSON (text area labeled Optional with placeholder text about overriding built-in recipes), and Security (checkbox for Use OpsWorks security groups set to Yes).

3. Configure your custom cookbooks.

The cookbook configuration options depend on which Chef version you have selected. For more information on Chef versions, see [Chef Versions \(p. 161\)](#). You must specify a custom cookbook repository for all Chef versions. For Chef 0.9 and 11.4 stacks, this repository must contain all of your cookbooks, which are then installed on every instance.

With Chef 11.10 stacks, you can use Berkshelf to manage your cookbooks. You can use Berkshelf to install cookbooks from multiple repositories, including AWS OpsWorks-compatible Chef community cookbooks. Chef 11.10 stacks have the following additional options:

- **Manage Berkshelf** – Specifies whether to use Berkshelf.
- **Berkshelf Version** – The Berkshelf version.

This is simply a text field, so make sure to enter a valid version. AWS OpsWorks supports Berkshelf versions 2.0.14, 3.0.1, and 3.1.1.

In addition to enabling Berkshelf and specifying the version, you must include a Berksfile in your custom cookbook repository's root directory that specifies the cookbooks and their repositories. The custom cookbook repository can also include custom cookbooks. For more information on how to use Berkshelf, see [Using Berkshelf \(p. 167\)](#).

When you are finished, click **Save** to save the updated stack.

**Important**

If you specify a custom cookbook repository, after you click **Save**, AWS OpsWorks automatically installs cookbooks on all new instances when they start. However, you must explicitly direct AWS OpsWorks to install these cookbooks on any existing instances by running the **update custom cookbooks** command. For more information, see [Updating Custom Cookbooks \(p. 174\)](#).

If you have enabled Berkshelf, the specified cookbooks are installed when you deploy the associated app.

## Specifying a Custom Cookbook Repository

If you want to use a custom cookbook repository, AWS OpsWorks can install the cookbooks from any of four repository types:

- HTTP archives are typically the best option for a publicly accessible archive, but can also be password protected.
- S3 archives are typically the preferred option for a private archive.
- Git and Subversion repositories provide source control and the ability to have multiple branches.

All repository types have the following required fields.

- **Repository type**—The repository type
- **Repository URL**—The repository URL

AWS OpsWorks supports publicly hosted Git repository sites such as [GitHub](#) or [Bitbucket](#) as well as privately hosted Git servers. For Git repositories, you must use one of the following URL formats, depending on whether the repository is public or private. Follow the same URL guidelines for Git submodules.

For a public repository, use the https or Git read-only protocols. For example, the following URLs are for the publicly available GitHub repository used in [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#):

- Git read-only: `git://github.com/amazonwebservices/opsworks-example-cookbooks.git`.
- HTTPS: `https://github.com/amazonwebservices/opsworks-example-cookbooks.git`.

For a private repository, you must use the SSH read/write format, as shown in the following examples:

- Github repositories: `git@github.com:project/repository`.
- Repositories on a Git server: `user@server:project/repository`

The remaining settings vary with the repository type and are described in the following sections.

## HTTP Archive

Selecting **Http Archive** for **Repository type** displays two additional settings, which you must complete if the archive is password protected.

- **User name**—Your user name
- **Password**—Your password

## Amazon S3 Archive

For private Amazon S3 archives, select **S3 Archive** for **Repository type**, which displays two additional required settings:

- **Access key ID**—Your AWS access key
- **Secret access key**—Your AWS secret key

For public Amazon S3 archives, specify **HTTP Archive** as the repository type.

## Git Repository

Selecting **Git** under **Source Control** displays are two additional optional settings:

### Repository SSH key

You must specify a deploy SSH key to access private Git repositories. For Git submodules, the specified key must have access to those submodules. For more information, see [Using Repository SSH Keys \(p. 146\)](#).

#### Important

The deploy SSH key cannot require a password; AWS OpsWorks has no way to pass it through.

### Branch/Revision

If the repository has multiple branches, AWS OpsWorks downloads the master branch by default. To specify a particular branch, enter the branch name, SHA1 hash, or tag name. To specify a particular commit, enter the full 40-hexdigit commit ID.

## Subversion Repository

Selecting **Subversion** under **Source Control** displays three additional settings:

- **User name**—Your user name, for private repositories.
- **Password**—Your password, for private repositories.
- **Branch/Revision**—[Optional] The revision name, if you have multiple revisions.

To specify a branch or tag, you must modify the repository URL, for example:`http://repository_domain/repos/myapp/branches/my-apps-branch` or `http://repository_domain_name/repos/calc/myapp/my-apps-tag`.

# Updating Custom Cookbooks

When you provide AWS OpsWorks with custom cookbooks, the built-in Setup recipes create a local cache on each newly started instance and download the cookbooks to the cache. AWS OpsWorks then runs recipes from the cache, not the repository. If you modify the custom cookbooks in the repository, you must ensure that the updated cookbooks are installed on your instances' local caches. AWS OpsWorks automatically deploys the latest cookbooks to new instances when they are started. For existing instances, however, the situation is different:

- You must manually deploy updated custom cookbooks to online instances.
- You do not have to deploy updated custom cookbooks to offline instance store-backed instances, including load-based and time-based instances.

AWS OpsWorks automatically deploys the current cookbooks when the instances are restarted.

- You must restart offline EBS-backed 24/7 instances and manually deploy your custom cookbooks.

AWS OpsWorks does not automatically update the custom cookbooks on these instances when they are restarted.

- You cannot restart offline EBS-backed load-based and time-based instances, so the simplest approach is to delete the offline instances and add new instances to replace them.

Because they are now new instances, AWS OpsWorks will automatically deploy the current custom cookbooks when the instances are started.

## To manually update custom cookbooks

1. Update your repository with the modified cookbooks. AWS OpsWorks uses the cache URL that you provided when you originally installed the cookbooks, so the cookbook root file name, repository location, and access rights should not change.
  - For Amazon S3 or HTTP repositories, replace the original .zip file with a new .zip file that has the same name.
  - For Git or Subversion repositories, [edit your stack settings \(p. 50\)](#) to change the **Branch/Revision** field to the new version.
2. On the stack's page, click **Run Command** and select the **Update Custom Cookbooks** command.

## Run Command

The screenshot shows the 'Run Command' configuration interface. At the top, there's a 'Settings' section with a dropdown menu set to 'Update Custom Cookbooks'. Below it is a 'Comment' field labeled 'Optional' with a text area. An 'Advanced »' link is located just below the comment field. The next section is 'Instances' with a note that OpsWorks will run the command on 1 of 3 instances. It lists three instance types: PHP App Server, MySQL, and db-master1. Under each type, there are checkboxes for selecting specific instances. The 'PHP App Server' section has checkboxes for 'PHP App Server' (checked), 'php-app1' (checked), and 'php-app2' (unchecked). The 'MySQL' section has checkboxes for 'MySQL' (unchecked) and 'db-master1' (checked). At the bottom right are 'Cancel' and 'Update Custom Cookbooks' buttons.

3. Add a comment if desired.
4. You can also add a custom JSON object to be merged into the stack configuration JSON that is passed to the instances (optional). For more information, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#) and [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).
5. By default, AWS OpsWorks updates the cookbooks on every instance. To specify which instances to update, select the appropriate instances from the list at the end of the page. To select every instance in a layer, select the appropriate layer checkbox in the left column.
6. Click **Update Custom Cookbooks** to install the updated cookbooks. AWS OpsWorks deletes the cached custom cookbooks on the specified instances and installs the new cookbooks from the repository. If you have enabled Berkshelf, **Update Custom Cookbooks** also updates your Berkshelf cookbooks. For more information on Berkshelf, see [Using Berkshelf \(p. 167\)](#).

### Note

This procedure is required only for existing instances, which have old versions of the cookbooks in their caches. If you subsequently add instances to a layer, AWS OpsWorks deploys the cookbooks that are currently in the repository so they automatically get the latest version.

# Executing Recipes

You can run recipes in two ways:

- Automatically, by assigning recipes to the appropriate layer's lifecycle event.
- Manually, by running the [Execute Recipes stack command \(p. 52\)](#) stack command or by using the agent CLI.

### Topics

- [AWS OpsWorks Lifecycle Events \(p. 176\)](#)
- [Automatically Running Recipes \(p. 177\)](#)
- [Manually Running Recipes \(p. 178\)](#)

## AWS OpsWorks Lifecycle Events

A layer has a sequence of five lifecycle events, each of which has an associated set of recipes that are specific to the layer. When an event occurs on a layer's instance, AWS OpsWorks automatically runs the appropriate set of recipes.

- **Setup** occurs on a new instance after it successfully boots. AWS OpsWorks runs recipes that set the instance up according to its layer. For example, if the instance is a member of the Rails App Server layer, the Setup recipes install Apache, Ruby Enterprise Edition, Passenger and Ruby on Rails. Setup includes Deploy, which automatically deploys the appropriate recipes to a new instance after setup is complete.
- **Configure** occurs on all of the stack's instances when an instance enters or leaves the online state. For example, suppose that your stack has instances A, B, and C, and you start a new instance, D. After D has finished running its setup recipes, AWS OpsWorks triggers the Configure event on A, B, C, and D. If you subsequently stop A, AWS OpsWorks triggers the Configure event on B, C, and D. AWS OpsWorks responds to the Configure event by running each layer's Configure recipes, which update the instances' configuration to reflect the current set of online instances. The Configure event is therefore a good time to regenerate configuration files. For example, the HAProxy Configure recipes reconfigure the load balancer to accommodate any changes in the set of online application server instances.
- **Deploy** occurs when you run a **Deploy** command, typically to deploy an application to a set of application server instances. The instances run recipes that deploy the application and any related files from its repository to the layer's instances. For example, for a Rails Application Server instances, the Deploy recipes check out a specified Ruby application and tell **Phusion Passenger** to reload it. You can also run **Deploy** on other instances so they can, for example, update their configuration to accommodate the newly deployed app. Note that Setup includes Deploy; it runs the Deploy recipes after setup is complete to automatically deploy the appropriate recipes to a new instance.
- **Undeploy** occurs when you delete an app or run an **Undeploy** command to remove an app from a set of application server instances. The specified instances run recipes to remove all application versions and perform any required cleanup.
- **Shutdown** occurs after you direct AWS OpsWorks to shut an instance down but before the associated Amazon EC2 instance is actually terminated. AWS OpsWorks runs recipes to perform cleanup tasks such as shutting down services. AWS OpsWorks allows Shutdown recipes 45 seconds to perform their tasks, and then terminates the Amazon EC2 instance.

For more discussion about the **Deploy** and **Undeploy** app commands, see [Deploying Apps \(p. 132\)](#).

After a new instance finishes booting, AWS OpsWorks does the following:

1. Runs the built-in Setup recipes.
2. Runs any custom Setup recipes.
3. Runs the built-in Deploy recipes.
4. Runs any custom Deploy recipes.

After AWS OpsWorks finishes executing these recipes and the new instance is online, AWS OpsWorks triggers a Configure event on all instances in the stack, including the new instance.

To respond to these events, implement custom recipes and assign them to the appropriate events for each layer. AWS OpsWorks runs those recipes after the event's built-in recipes.

### Tip

To see the lifecycle events that have occurred on a particular instance, go to the **Instances** page and click the instance's name to open its details page. The list of events is in the **Logs** section at the bottom of the page. You can click **show** in the **Log** column to examine the Chef log for an event, which provides detailed information about how the event was handled, including which recipes were run. For more information on how to interpret Chef logs, see [Chef Logs \(p. 338\)](#).

SSH Fingerprint: 22:1c:6e:e1:4d:40:a9:0f:2a:4c:c7:1c:50:00			
Logs			
Created at	Command	Duration	Log
✓ 2014-02-21 17:25:13 UTC	shutdown	00:00:28	<a href="#">show</a>
✓ 2014-02-21 17:09:21 UTC	configure	00:01:04	<a href="#">show</a>
✓ 2014-02-21 17:08:35 UTC	setup	00:00:45	<a href="#">show</a>
✓ 2014-02-21 17:03:34 UTC	shutdown	00:01:06	<a href="#">show</a>
✓ 2014-02-17 21:17:49 UTC	configure	00:01:05	<a href="#">show</a>
✓ 2014-02-17 21:11:15 UTC	setup	00:06:33	<a href="#">show</a>

For each lifecycle event, AWS OpsWorks provides each instance with a [stack configuration and deployment JSON \(p. 262\)](#) that contains the current stack state and, for Deploy events, a variety of information about the deployment. This JSON includes information about what instances are available, their IP addresses, and so on. For more information, see [Stack Configuration and Deployment JSON \(p. 262\)](#).

**Note**

Starting or stopping a large number of instances at the same time can rapidly generate a large number of Configure events. To avoid unnecessary processing, AWS OpsWorks responds to only the last event. That event's stack configuration and deployment JSON contains all the information required to update the stack's instances for the entire set of changes. This eliminates the need to also process the earlier Configure events. AWS OpsWorks labels the unprocessed Configure events as **superseded**.

## Automatically Running Recipes

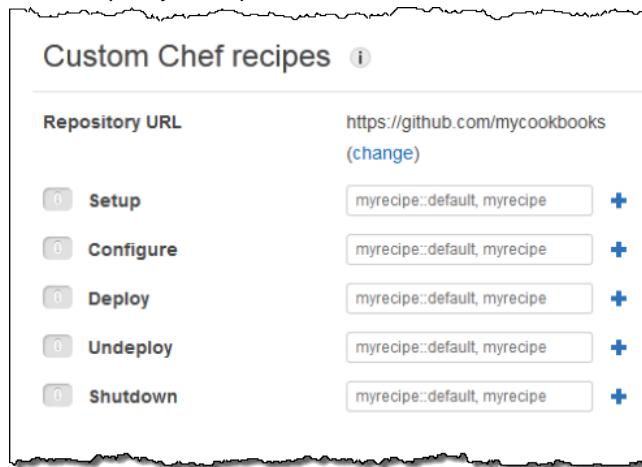
Each layer has a set of built-in recipes assigned to each lifecycle event, although some layers lack Undeploy recipes. When a lifecycle event occurs on an instance, AWS OpsWorks runs the appropriate set of recipes for the associated layer. To see which built-in recipes are assigned to a layer's events, open the layer's **Recipes** tab and look under **Built-in Chef recipes**.

Built-in Chef recipes	
We have defined 20 built-in Chef recipes for this layer. <a href="#">Hide</a> <	
<b>10</b>	<b>Setup</b>
	<a href="#">opsworks_initial_setup</a> <a href="#">ssh_host_keys</a> <a href="#">ssh_users</a> <a href="#">mysql:client</a> <a href="#">dependencies</a> <a href="#">ebs</a>
	<a href="#">opsworks_ganglia:client</a> <a href="#">mysql:client</a> <a href="#">dependencies</a> <a href="#">mod_php5_apache2</a>
<b>5</b>	<b>Configure</b>
	<a href="#">opsworks_ganglia:configure-client</a> <a href="#">ssh_users</a> <a href="#">agent_version</a> <a href="#">mod_php5_apache2:php</a> <a href="#">php:configure</a>
<b>2</b>	<b>Deploy</b>
	<a href="#">deploy:default</a> <a href="#">deploy:php</a>
<b>1</b>	<b>Undeploy</b>
	<a href="#">deploy:php-undeploy</a>
<b>2</b>	<b>Shutdown</b>
	<a href="#">opsworks_shutdown:default</a> <a href="#">apache2:stop</a>

If you have installed custom cookbooks, you can have AWS OpsWorks run some or all of the recipes automatically by assigning each recipe to a layer's lifecycle event. After an event occurs, AWS OpsWorks runs the specified custom recipes after the layer's built-in recipes.

### To assign custom recipes to layer events

1. On the **Layers** page, for the appropriate layer, click **Recipes** and then click **Edit**. If you haven't yet enabled custom cookbooks, click **configure cookbooks** to open the stack's **Settings** page. Toggle **Use custom Chef Cookbooks** to **Yes**, and provide the cookbook's repository information. Then click **Save** and navigate back to the edit page for the **Recipes** tab. For more information, see [Installing Custom Cookbooks \(p. 171\)](#).
2. On the **Recipes** tab, enter each custom recipe in the appropriate event field and click **+** to add it to the list. Specify a recipe as follows: `cookbook::somerecipe` (omit the `.rb` extension).



When you start a new instance, AWS OpsWorks automatically runs the custom recipes for each event, after it runs the standard recipes.

#### Note

Custom recipes execute in the order that you enter them in the console. An alternative way to control execution order is to implement a meta recipe that executes the recipes in the correct order. For more information, see [Recipes \(p. 159\)](#).

## Manually Running Recipes

Although recipes are typically run automatically in response to lifecycle events, you can manually run recipes at any time on any or all stack instances. This feature is typically used for tasks that don't naturally map to a lifecycle event, such backing up instances. To run a custom recipe manually, it must be in one of your custom cookbooks but does not have to be assigned to a lifecycle event. When you run a recipe manually, AWS OpsWorks includes the same deployment attributes in the [stack configuration and deployment JSON \(p. 262\)](#) that it does for a Deploy event.

### To manually run recipes on stack instances

1. On the **Stack** page, click **Run command**. For **Command**, select **Execute Recipes**.

## Run Command

Settings

Command	Execute Recipes
Recipes to execute	<input type="text"/>
Comment	<input type="text"/> Optional
Custom Chef JSON	<input type="text"/> Optional

Enter custom JSON that is passed to your Chef recipes for all instances in your stack. You can use this to override and customize built-in recipes or pass variables to your own. [Learn more](#).

Instances i

No running instances with the OpsWorks status online or setup\_failed. Start [instances](#) now.

[Cancel](#) [Execute Recipes](#)

2. Enter the recipes to be run in the **Recipes to execute** box by using the standard `cookbookname::recipename` format. Use commas to separate multiple recipes; they will run in the order that you list them.
3. Optionally, use the **Custom Chef JSON** box to add a custom JSON object that will be merged into the stack configuration JSON that is passed to the instances. For more information about using how custom JSON objects, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#) and [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).
4. Under **Instances**, select the instances on which AWS OpsWorks should run the recipes.

When a lifecycle event occurs, the AWS OpsWorks receives a command to run the associated recipes. You can manually run these commands on a particular instance by using the agent CLI's [run\\_command \(p. 373\)](#) command. This command is particularly useful for testing. For example, the Setup event normally occurs only once, after an instance boots. If you want to test custom Setup recipes, it is useful to be able to rerun them without having to take the time required to stop and restart the instance. You can instead [use SSH \(p. 119\)](#) to log into the instance and run the [run\\_command \(p. 373\)](#) command to rerun the Setup recipes as required. For more information on how to use the agent CLI, see [Appendix B: Instance Agent CLI \(p. 368\)](#).

## Cookbooks 101

A production-level AWS OpsWorks stack typically requires some [customization \(p. 230\)](#), which often means implementing a custom Chef cookbook with one or more custom recipes, attribute files, or template files. This chapter is a general introduction to implementing cookbooks for AWS OpsWorks. It assumes that you have already read [Cookbooks and Recipes \(p. 153\)](#) and are generally familiar with cookbooks and how they are used by AWS OpsWorks. For additional information on how to implement and test Chef recipes, see [Test-Driven Infrastructure with Chef, 2nd Edition](#).

The example walkthroughs that follow show the basics of how to implement cookbooks to perform common tasks, such as installing packages or creating directories. To simplify the process, you will use a pair of useful tools to run most of the examples locally in a virtual machine.

- [Vagrant](#) simplifies the process of using a virtual machine to run instances on your local system.
- [Test Kitchen](#) simplifies the process of executing and testing recipes.

Most of the examples use Test Kitchen to run your recipes locally on Vagrant, but final section shows how to use Test Kitchen to run recipes on an Amazon EC2 instance.

#### Topics

- [Vagrant and Test Kitchen \(p. 180\)](#)
- [Cookbook Basics \(p. 181\)](#)
- [Implementing Cookbooks for AWS OpsWorks \(p. 209\)](#)

## Vagrant and Test Kitchen

This topic provides brief descriptions of Vagrant and Test Kitchen, and points you to installation instructions and walkthroughs that will get you set up and familiarize you with the basics of how to use the tools.

#### Topics

- [Vagrant \(p. 180\)](#)
- [Test Kitchen \(p. 180\)](#)

## Vagrant

[Vagrant](#) provides a consistent environment for executing and testing code on a virtual machine. It supports a wide variety of environments—called Vagrant boxes—each of which represents a configured operating system. For AWS OpsWorks, the environments of interest are based on Ubuntu or Amazon Linux, so the examples primarily use a Vagrant box named `opscode-ubuntu-12.04`. It represents a Ubuntu 12.04 LTS system that is configured for Chef.

Vagrant is available for Linux, Windows, and Macintosh systems, so you can use your preferred workstation to implement and test recipes on any supported operating system. The examples for this chapter were created on an Ubuntu 12.04 LTS Linux system, but translating the procedures to Windows or Macintosh systems is straightforward.

Vagrant is basically a wrapper for a virtualization provider. Most of the examples use the [VirtualBox](#) provider. VirtualBox is free and available for Linux, Windows, and Macintosh systems. The Vagrant walkthrough provides installation instructions if you do not already have VirtualBox on your system. Note that you can run Ubuntu-based environments on VirtualBox, but Amazon Linux is available only for Amazon EC2 instances. However, you can run a similar operating system such as CentOS on VirtualBox, which is useful for initial development and testing.

For information on other providers, see the [Vagrant](#) documentation. In particular, the `vagrant-aws` plugin provider allows you to use Vagrant with Amazon EC2 instances. This provider is particularly useful for testing recipes on Amazon Linux, which is available only on Amazon EC2 instances. The `vagrant-aws` provider is free, but you must have an AWS account and pay for any AWS resources that you use.

At this point, you should go through Vagrant's [Getting Started walkthrough](#), which describes how to install Vagrant on your workstation and teaches you the basics of how to use Vagrant. Note that the examples in this chapter do not use a Git repository, so you can omit that part of the walkthrough if you prefer.

## Test Kitchen

[Test Kitchen](#) simplifies the process of executing and testing your cookbooks on Vagrant. As a practical matter, you rarely if ever need to use Vagrant directly. Test Kitchen performs most common tasks, including:

- Launching an instance in Vagrant.
- Transferring cookbooks to the instance.
- Running the cookbook's recipes on the instance.
- Testing a cookbook's recipes on the instance.
- Using SSH to log in to the instance.

Instead of installing the Test Kitchen gem directly, we recommend installing [Chef DK](#). In addition to Chef itself, this package includes Test Kitchen, [Berkshelf](#), [ChefSpec](#), and several other useful tools.

At this point, you should go through Test Kitchen's [Getting Started walkthrough](#), which teaches you the basics of how to use Test Kitchen to execute and test recipes.

**Tip**

The examples in this chapter use Test Kitchen as a convenient way to run recipes. If you prefer, you can stop the Getting Started walkthrough after completing the Manually Verifying section, which covers everything you need to know for the examples. However, Test Kitchen is primarily a testing platform that supports test frameworks such as [bash automated test system \(BATS\)](#). You should complete the remainder of the walkthrough at some point to learn how to use Test Kitchen to test your recipes.

## Cookbook Basics

You can use cookbooks to accomplish a wide variety of tasks. The following topics assume you are new to Chef, and describe how to use cookbooks to accomplish some common asks.

**Note**

Before continuing, make sure that you have installed Vagrant and Test Kitchen, and gone through their Getting Started walkthroughs. For more information, see [Vagrant and Test Kitchen \(p. 180\)](#).

**Topics**

- [Recipe Structure \(p. 181\)](#)
- [Example 1: Installing Packages \(p. 184\)](#)
- [Example 2: Managing Users \(p. 186\)](#)
- [Example 3: Creating Directories \(p. 187\)](#)
- [Example 4: Adding Flow Control \(p. 189\)](#)
- [Example 5: Using Attributes \(p. 192\)](#)
- [Example 6: Creating Files \(p. 195\)](#)
- [Example 7: Running Commands and Scripts \(p. 199\)](#)
- [Example 8: Managing Services \(p. 201\)](#)
- [Example 9: Using Amazon EC2 Instances \(p. 205\)](#)
- [Next Steps \(p. 209\)](#)

## Recipe Structure

A cookbook is primarily a set of *recipes*, which can perform a wide variety of tasks on an instance, but it can also include a variety of supporting files such as template and attribute files. To clarify how to implement recipes, it's useful to look at a simple example. The following is the setup recipe for the built-in [HAProxy layer \(p. 72\)](#). Just focus on the overall structure at this point and don't worry too much about the details; they will be covered in the subsequent examples.

```
package 'haproxy' do
```

```
action :install
end

if platform?('debian', 'ubuntu')
  template '/etc/default/haproxy' do
    source 'haproxy-default.erb'
    owner 'root'
    group 'root'
    mode 0644
  end
end

include_recipe 'haproxy::service'

service 'haproxy' do
  action [:enable, :start]
end

template '/etc/haproxy/haproxy.cfg' do
  source 'haproxy.cfg.erb'
  owner 'root'
  group 'root'
  mode 0644
  notifies :restart, "service[haproxy]"
end
```

**Tip**

For this and other examples of working recipes and related files, see the [AWS OpsWorks built-in recipes](#).

The example highlights the key recipe elements, which are described in the following sections.

**Topics**

- [Resources \(p. 182\)](#)
- [Flow Control \(p. 183\)](#)
- [Included Recipes \(p. 184\)](#)

## Resources

Recipes consist largely of a set of Chef *resources*. Each one specifies a particular aspect of the instance's final state, such as a package to be installed or a service to be started. The example has four resources:

- A `package` resource, which represents an installed package, an [HAProxy server](#) for this example.
- A `service` resource, which represents a service, the HAProxy service for this example.
- Two `template` resources, which represent files that are to be created from a specified template, two HAProxy configuration files for this example.

Resources provide a declarative way to specify the instance state. Behind the scenes, each resource has an associated *provider* that performs the required tasks, such as installing packages, creating and configuring directories, starting services, and so on. If the details of the task depend on the particular operating system, the resource has multiple providers and uses the appropriate one for the system. For example, on a Red Hat Linux system the `package` provider uses `yum` to install packages. On a Ubuntu Linux system, the `package` provider uses `apt-get`.

You implement a resource as a Ruby code block with the following general format.

```
resource_type "resource_name" do
  attribute1 'value1'
  attribute2 'value2'
  ...
  action :action_name
  notifies :action, 'resource'
end
```

The elements are:

#### Resource type

(Required) The example includes three resource types, package, service, and template.

#### Resource name

(Required) The name identifies the particular resource and is sometimes used as a default value for one of the attributes. In the example, package represents a package resource named haproxy and the first template resource represents a configuration file named /etc/default/haproxy.

#### Attributes

(Optional) Attributes specify the resource configuration and vary depending on the resource type.

- The example's template resources explicitly define a set of attributes that specify the created file's source, owner, group, and mode.
- The example's package and service resources do not explicitly define any attributes.

The resource name is typically the default value for a required attribute and is sometimes all that is needed. For example, the resource name is the default value for the package resource's package\_name attribute, which is the only required attribute.

There are also some specialized attributes called guard attributes, which specify when the resource provider is to take action. For example, the only\_if attribute directs the resource provider to take action only if a specified condition is met. The HAProxy recipe does not use guard attributes, but they are used by several of the following examples.

#### Actions and Notifications

(Optional) Actions and notifications specify what tasks the provider is to perform.

- action directs the provider to take a specified action, such as install or create.

Each resource has a set of actions that depend on the particular resource, one of which is the default action. In the example, the package resource's action is install, which directs the provider to install the package. The first template resource has no action element, so the provider takes the default create action.

- notifies directs another resource's provider to perform an action, but only if the resource's state has changed.

notifies is typically used with resources such as template and file to perform tasks such as restarting a service after modifying a configuration file. Resources do not have default notifications. If you want a notification, the resource must have an explicit notifies element. In the HAProxy recipe, the second template resource notifies the haproxy service resource to restart the HAProxy service if the associated configuration file has changed.

For descriptions of the standard resources, including the available attributes, actions, and notifications for each resource, see [About Resources and Providers](#).

## Flow Control

Because recipes are Ruby applications, you can use Ruby control structures to incorporate flow control into a recipe. For example, you can use Ruby conditional logic to have the recipe behave differently on

different systems. The HAProxy recipe includes an `if` block that uses a `template` resource to create a configuration file, but only if the recipe is running on a Debian or Ubuntu system.

Another common scenario is using a loop to execute a resource multiple times with different attribute settings. For example, you can create a set of directories by using a loop to execute a `directory` resource multiple times with different directory names.

**Tip**

If you aren't familiar with Ruby, see [Just Enough Ruby for Chef](#), which covers what you need to know for most recipes.

## Included Recipes

`include_recipe` includes other recipes in your code, which allows you to modularize your recipes and reuse the same code in multiple recipes. When you run the host recipe, Chef replaces each `include_recipe` element with the specified recipe's code before it executes the host recipe. You identify an included recipe by using the standard Chef `cookbook_name::recipe_name` syntax, where `recipe_name` omits the `.rb` extension. The example includes one recipe, `haproxy::service`, which represents the HAProxy service.

**Note**

If you use `include_recipe` in recipes running on Chef 11.10 and later to include a recipe from another cookbook, you must use a `depends` statement to declare the dependency in the cookbook's `metadata.rb` file. For more information, see [Implementing Recipes for Chef 11.10 Stacks \(p. 163\)](#).

## Example 1: Installing Packages

Package installation is one of the more common uses of recipes and can be quite simple, depending on the package. For example, the following recipe installs Git.

```
package 'git' do
  action :install
end
```

The `package` resource handles package installation. For this example, you don't need to specify any attributes. The resource name is the default value for the `package_name` attribute, which identifies the package. The `install` action directs the provider to install the package. You could make the code even simpler by skipping `install`; it's the `package` resource's default action. When you run the recipe, Chef uses the appropriate provider to install the package. On the Ubuntu system that you will use for the example, the provider installs Git by calling `apt-get`.

To use Test Kitchen to run this recipe in Vagrant, you first need to set up a cookbook and initialize and configure Test Kitchen. The following is for a Linux system, but the procedure is essentially similar for Windows and Macintosh systems. Start by opening a Terminal window; all of the examples in this chapter use command-line tools.

### To prepare the cookbook

1. In your home directory, create a subdirectory named `opsworks_cookbooks`, which will contain all the cookbooks for this chapter. Then create a subdirectory for this cookbook named `installpkg` and navigate to it.
2. In `installpkg`, create a file named `metadata.rb` that contains the following code.

```
name "installpkg"
version "0.1.0"
```

For simplicity, the examples in this chapter just specify the cookbook name and version, but `metadata.rb` can contain a variety of cookbook metadata. For more information, see [About Cookbook Metadata](#).

**Tip**

Make sure to create `metadata.rb` before you initialize Test Kitchen; it uses the data to create the default configuration file.

3. In `installpkg`, run `kitchen init`, which initializes Test Kitchen and installs the default Vagrant driver.
4. The `kitchen init` command creates a YAML configuration file in `installpkg` named `.kitchen.yml`. Open the file in your favorite text editor. The `.kitchen.yml` file includes a `platforms` section that specifies which systems to run the recipes on. Test Kitchen creates an instance and runs the specified recipes on each platform.

**Tip**

By default, Test Kitchen runs recipes one platform at a time. If you add a `-p` argument to any command that creates an instance, Test Kitchen will run the recipes on every platform, in parallel.

A single platform is sufficient for this example, so edit `.kitchen.yml` to remove the `centos-6.4` platform. Your `.kitchen.yml` file should now look like this:

```
---  
driver:  
  name: vagrant  
  
provisioner:  
  name: chef_solo  
  
platforms:  
  - name: ubuntu-12.04  
  
suites:  
  - name: default  
    run_list:  
      - recipe[installpkg::default]  
    attributes:
```

Test Kitchen runs only those recipes that are in the `.kitchen.yml` run list. You identify recipes by using the `[cookbook_name>::recipe_name]` format, where `recipe_name` omits the `.rb` extension. Initially, the `.kitchen.yml` run list contains the cookbook's default recipe, `installpkg::default`. That's the recipe that you are going to implement, so you don't need to modify the run list.

5. Create a subdirectory of `installpkg` named `recipes`.

If a cookbook contains recipes—most do—they must be in the `recipes` subdirectory.

You can now add the recipe to the cookbook and use Test Kitchen to run it on an instance.

### To run the recipe

1. Create a file named `default.rb` that contains the Git installation example code from the beginning of the section and save it to the `recipes` subdirectory.
2. In the `installpkg` directory, run `kitchen converge`. This command starts a new Ubuntu 12.04 LTS instance in Vagrant, copies your cookbooks to the instance, and initiates a Chef run to execute the recipes in the `.kitchen.yml` run list.

3. To verify that the recipe was successful, run `kitchen login`, which opens an SSH connection to the instance. Then run `git --version` to verify that Git was successfully installed. To return to your workstation, run `exit`.
4. When you are finished, run `kitchen destroy` to shut down the instance. The next example uses a different cookbook.

This example was a good way to get started, but it is especially simple. Other packages can be more complicated to install; you might need to do any or all of the following:

- Create and configure a user.
- Create one or more directories for data, logs, and so on.
- Install one or more configuration files.
- Specify a different package name or attribute values for different operating systems.
- Start a service and then restart it as needed.

The following examples describe how to address these issues, along with some other useful operations.

## Example 2: Managing Users

Another simple task is managing users on an instance. The following recipe adds a new user to an instance.

```
user "myuser" do
  home "/home/newuser"
  shell "/bin/bash"
end
```

You use a `user` resource to manage users. The example creates a user named `myuser` and specifies their home directory and shell. There is no action specified, so the resource uses the default `create` action. You can add attributes to `user` to specify a variety of other settings, such as their password or group ID. You can also use `user` for related user-management tasks such as modifying user settings or deleting users. For more information, see [user](#).

### To run the recipe

1. Create a directory within `opsworks_cookbooks` named `newuser` and navigate to it.
2. Create a `metadata.rb` file that contains the following code and save it to `newuser`.

```
name "newuser"
version "0.1.0"
```

3. Initialize and configure Test Kitchen, as described in [Example 1: Installing Packages \(p. 184\)](#), and add a `recipes` directory inside the `newuser` directory.
4. Add `default.rb` file with the example recipe to the cookbook's `recipes` directory .
5. Run `kitchen converge` to execute the recipe.
6. Use `kitchen login` to log in to the instance and verify the new user's existence by running `cat /etc/passwd`. The `myuser` user should be at the bottom of the file.

## Example 3: Creating Directories

When you install a package on an instance, you often need to create some configuration files and place them in the appropriate directories. However, those directories might not exist yet. You might also need to create directories for data, log files, and so on. For example, you first boot the Ubuntu 12.04 LTS system that you use for most of the examples, the `/srv` directory has no subdirectories. If you are installing an application server, you will probably want a `/srv/www/` directory and perhaps some subdirectories for data files, logs, and so on. The following recipe creates `/srv/www/` on an instance.

```
directory "/srv/www/" do
  mode 0755
  owner 'root'
  group 'root'
  action :create
end
```

You use a `directory` resource to create and configure directories. The resource name is the default value for the resource's `path` attribute, so the example creates `/srv/www/` and specifies its `mode`, `owner`, and `group` properties.

### To run the recipe

1. Create a directory inside `opsworks_cookbooks` named `createdir` and navigate to it.
2. Initialize and configure Test Kitchen, as described in [Example 1: Installing Packages \(p. 184\)](#), and add a `recipes` directory within `createdir`.
3. Add a `default.rb` file with the recipe code to the cookbook's `recipes` subdirectory.
4. Run `kitchen converge` to execute the recipe.
5. Run `kitchen login`, navigate to `/srv` and verify that it has a `www` subdirectory.
6. Run `exit` to return to your workstation but leave the instance running.

### Tip

To create a directory relative to your home directory on the instance, use `#{$ENV['HOME']}` to represent the home directory. For example, the following creates the `~/shared` directory.

```
directory "#{$ENV['HOME']}/shared" do
  ...
end
```

Suppose that you want to create a more deeply nested directory, such as `/srv/www/shared`. You could modify the preceding recipe as follows.

```
directory "/srv/www/shared" do
  mode 0755
  owner 'root'
  group 'root'
  action :create
end
```

## To run the recipe

1. Replace the code in `default.rb` with the preceding recipe.
2. Run `kitchen converge` from the `createdir` directory.
3. To verify that the directory was indeed created, run `kitchen login`, navigate to `/srv/www`, and verify that it contains a `shared` subdirectory.
4. Run `kitchen destroy` to shut the instance down.

You will notice the `kitchen converge` command ran much faster. That's because the instance is already running, so there's no need to boot the instance, install Chef, and so on. Test Kitchen just copies the updated cookbook to the instance and starts a Chef run.

Now run `kitchen converge` again, which executes the recipe on a fresh instance. You'll now see the following result.

```
Chef Client failed. 0 resources updated in 1.908125788 seconds
[2014-06-20T20:54:26+00:00] ERROR: directory[/srv/www/shared] (createdir::default
line 1) had an error: Chef::Exceptions::EnclosingDirectoryDoesNotExist: Parent
directory /srv/www does not exist, cannot create /srv/www/shared
[2014-06-20T20:54:26+00:00] FATAL: Chef::Exceptions::ChildConvergeError: Chef
run process exited unsuccessfully (exit code 1)
>>>>> Converge failed on instance <default-ubuntu-1204>.
>>>>> Please see .kitchen/logs/default-ubuntu-1204.log for more details
>>>>> -----Exception-----
>>>>> Class: Kitchen::ActionFailed
>>>>> Message: SSH exited (1) for command: [sudo -E chef-solo --config
/tmp/kitchen/solo.rb --json-attributes /tmp/kitchen/dna.json --log_level info]
>>>>> -----
```

What happened? The problem is that by default, a `directory` resource can create only one directory at a time; it can't create a chain of directories. The reason the recipe worked earlier is that the very first recipe you ran on the instance had already created `/srv/www`, so creating `/srv/www/shared` created only one subdirectory.

### Tip

When you run `kitchen converge`, make sure you know whether you are running your recipes on a new or existing instance. You might get different results.

To create a chain of subdirectories, add a `recursive` attribute to `directory` and set it to `true`. The following recipe creates `/srv/www/shared` directly on a clean instance.

```
directory "/srv/www/shared" do
  mode 0755
  owner 'root'
  group 'root'
  recursive true
  action :create
end
```

## Example 4: Adding Flow Control

Some recipes are just a series of Chef resources. In that case, when you run the recipe, it simply executes each of the resource providers in sequence. However, it's often useful to have a more sophisticated execution path. The following are two common scenarios:

- You want a recipe to execute the same resource multiple times with different attribute settings.
- You want to use different attribute settings on different operating systems.

You can address scenarios such as these by incorporating Ruby control structures into the recipe. This section shows how to modify the recipe from [Example 3: Creating Directories \(p. 187\)](#) to address both scenarios.

### Topics

- [Iteration \(p. 189\)](#)
- [Conditional Logic \(p. 190\)](#)

## Iteration

[Example 3: Creating Directories \(p. 187\)](#) showed how to use a directory resource to create a directory or chain of directories. However, suppose that you want to create two separate directories, /srv/www/config and /srv/www/shared. You could implement a separate directory resource for each directory, but that approach can get cumbersome if you want to create very many directories. The following recipe shows a simpler way to handle the task.

```
[ "/srv/www/config", "/srv/www/shared" ].each do |path|
  directory path do
    mode 0755
    owner 'root'
    group 'root'
    recursive true
    action :create
  end
end
```

Instead of using a separate directory resource for each subdirectory, the recipe uses a string collection that contains the subdirectory paths. The Ruby each method executes the resource once for each collection element, starting with the first one. The element's value is represented in the resource by the path variable, which in this case represents the directory path. You can easily adapt this example to create any number of subdirectories.

### To run the recipe

1. Stay in createdir directory; you'll be using that cookbook for the next several examples.
2. If you haven't done so already, run `kitchen destroy` so you are starting with a clean instance.
3. Replace the code in `default.rb` with the example and run `kitchen converge`.
4. Log in to the instance; you will see the newly created directories under `/srv`.

You can use a hash table to specify two values for each iteration. The following recipe creates /srv/www/config and /srv/www/shared, each with a different mode.

```
{ "/srv/www/config" => 0644, "/srv/www/shared" => 0755 }.each do |path,  
  mode_value|  
  directory path do  
    mode mode_value  
    owner 'root'  
    group 'root'  
    recursive true  
    action :create  
  end  
end
```

### To run the recipe

1. If you haven't done so already, run `kitchen destroy` so you are starting with a clean instance.
2. Replace the code in `default.rb` with the example and run `kitchen converge`.
3. Log in to the instance; you will see the newly created directories under `/srv` with the specified modes.

#### Tip

AWS OpsWorks recipes commonly use this approach to extract values from the [stack configuration and deployment JSON \(p. 262\)](#)—which is basically a large hash table—and insert them in a resource. For an example, see [Deploy Recipes \(p. 254\)](#).

## Conditional Logic

You can also use Ruby conditional logic to create multiple execution branches. The following recipe uses `if-elsif-else` logic to extend the previous example so that it creates a subdirectory named `/srv/www/shared`, but only on Debian and Ubuntu systems. For all other systems, it logs an error message that is displayed in the Test Kitchen output.

```
if platform?("debian", "ubuntu")  
  directory "/srv/www/shared" do  
    mode 0755  
    owner 'root'  
    group 'root'  
    recursive true  
    action :create  
  end  
else  
  log "Unsupported system"  
end
```

### To run the example recipe

1. If your instance is still up, run `kitchen destroy` to shut it down.
2. Replace the code in `default.rb` with the example code.
3. Edit `.kitchen.yml` to add a CentOS 6.4 system to the platform list. The file's `platforms` section should now look like.

```
...  
platforms:  
  - name: ubuntu-12.04
```

```
- name: centos-6.4
...
```

- Run `kitchen converge`, which will create an instance and run the recipes for each platform in `.kitchen.yml`, in sequence.

**Tip**

If you want to converge just one instance, add the instance name as a parameter. For example, to converge the recipe only on the Ubuntu platform, run `kitchen converge default-ubuntu-1204`. If you forget the platform names, just run `kitchen list`.

You should see your log message in the CentOS part of the Test Kitchen output, which will look something like the following:

```
...
Converging 1 resources
Recipe: createdir::default
* log[Unsupported system] action write[2014-06-23T19:10:30+00:00] INFO: Processing log[Unsupported system] action write (createdir::default line 12)
[2014-06-23T19:10:30+00:00] INFO: Unsupported system

[2014-06-23T19:10:30+00:00] INFO: Chef Run complete in 0.004972162 seconds
```

You can now log in to the instances and verify that the directories were or were not created. However, you can't simply run `kitchen login now`. You must specify which instance by appending the platform name, for example, `kitchen login default-ubuntu-1204`.

**Tip**

If a Test Kitchen command takes an instance name, you don't need to type the complete name. Test Kitchen treats an instance name as a Ruby regular expression, so you just need enough characters to provide a unique match. For example, you can converge just the Ubuntu instance by running `kitchen converge ub` or log in to the CentOS instance by running `kitchen login 64`.

The question you probably have at this point is how the recipe knows which platform it is running on. Chef runs a tool called `Ohai` for every run that collects system data, including the platform, and represents it as a set of attributes in a structure called the *node object*. The `Chef.platform?` method compares the systems in parentheses against the Ohai platform value, and returns true if one of them matches.

You can reference the value of a node attribute directly in your code by using `node['attribute_name']`. The platform value, for example, is represented by `node['platform']`. You could, for example, have written the preceding example as follows.

```
if node[:platform] == 'debian' or node[:platform] == 'ubuntu'
  directory "/srv/www/shared" do
    mode 0755
    owner 'root'
    group 'root'
    recursive true
    action :create
  end
else
```

```
    log "Unsupported system"
end
```

A common reason for including conditional logic in a recipe is to accommodate the fact that different Linux families sometimes use different names for packages, directories, and so on. For example, the Apache package name is `httpd` on CentOS systems and `apache2` on Ubuntu systems.

If you just need a different string for different systems, the Chef `value_for_platform` method is a simpler solution than `if-elsif-else`. The following recipe creates a `/srv/www/shared` directory on CentOS systems, a `/srv/www/data` directory on Ubuntu systems, and `/srv/www/config` on all others.

```
data_dir = value_for_platform(
  "centos" => { "default" => "/srv/www/shared" },
  "ubuntu" => { "default" => "/srv/www/data" },
  "default" => "/srv/www/config"
)
directory data_dir do
  mode 0755
  owner 'root'
  group 'root'
  recursive true
  action :create
end
```

`value_for_platform` assigns the appropriate path to `data_dir` and the `directory` resource uses that value to create the directory.

### To run the example recipe

1. If your instance is still up, run `kitchen destroy` to shut it down.
2. Replace the code in `default.rb` with the example code.
3. Run `kitchen converge` and then login to each instance to verify that the appropriate directories are present.

## Example 5: Using Attributes

The recipes in the preceding sections used hard-coded values for everything other than the platform. This approach can be inconvenient if, for example, you want to use the same value in more than one recipe. You can define values separately from recipes by including an attribute file in your cookbook.

An attribute file is a Ruby application that assigns values to one or more attributes. It must be in the cookbook's `attributes` folder. Chef incorporates the attributes into the node object and any recipe can use the attribute values by referencing the attribute. This topic shows how to modify the recipe from [Iteration \(p. 189\)](#) to use attributes. Here's the original recipe for reference.

```
[ "/srv/www/config", "/srv/www/shared" ].each do |path|
  directory path do
    mode 0755
    owner 'root'
    group 'root'
    recursive true
    action :create
```

```
    end  
end
```

The following defines attributes for the subdirectory name, mode, owner, and group values.

```
default['createdir']['shared_dir'] = 'shared'  
default['createdir']['config_dir'] = 'config'  
default['createdir']['mode'] = 0755  
default['createdir']['owner'] = 'root'  
default['createdir']['group'] = 'root'
```

Note the following:

- Each definition starts with an *attribute type*.

If an attribute is defined more than once—perhaps in different attribute files—the attribute type specifies the attribute's precedence, which determines which definition is incorporated into the node object. For more information, see [Attribute Precedence \(p. 232\)](#). All the definitions in this example have the `default` attribute type, which is the usual type for this purpose.

- The attributes have nested names.

The node object is basically a hash table that can be nested arbitrarily deeply, so attribute names can be and commonly are nested. This attribute file follows a standard practice of using a nested name with the cookbook name, `createdir`, as the first element.

The reason for using `createdir` as the attribute's first element is that when you do a Chef run, Chef incorporates the attributes from every cookbook into the node object. With AWS OpsWorks, the node object includes a large number of attributes from the [built-in cookbooks](#) in addition to any attributes that you define. Including the cookbook name in the attribute name reduces the risk of a name collision with attributes from another cookbook, especially if your attribute has a name like `port` or `user`. Don't name an attribute something like [\[:apache2\]\[:user\] \(p. 405\)](#), for example, unless you want to override that attribute's value. For more information, see [Overriding AWS OpsWorks Attributes Using Custom Cookbook Attributes \(p. 235\)](#).

The following example shows the original recipe using attributes instead of hard-coded values.

```
[ "/srv/www/#{node['createdir']['shared_dir']}", "/srv/www/#{node['cre  
atedir']['config_dir']}"].each do |path|  
  directory path do  
    mode node['createdir']['mode']  
    owner node['createdir']['owner']  
    group node['createdir']['group']  
    recursive true  
    action :create  
  end  
end
```

### Tip

If you want to incorporate an attribute value into a string, wrap it with `#{}.` In the preceding example, `#{{node['createdir']['shared_dir']}} appends "shared" to "/srv/www/"`.

## To run the recipe

1. Run `kitchen destroy` to start with a clean instance.
2. Replace the code in `recipes/default.rb` with the preceding recipe example.
3. Create a subdirectory of `createdir` named `attributes` and add a file named `default.rb` that contains the attribute definitions.
4. Edit `.kitchen.yml` to remove CentOS from the platforms list.
5. Run `kitchen converge` and then log in to the instance and verify that `/srv/www/shared` and `/srv/www/shared` are there.

### Tip

With AWS OpsWorks, defining values as attributes provides an additional benefit; you can use [custom JSON](#) to override those values on a per-stack or even per-deployment basis. This can be useful for a variety of purposes, including the following:

- You can customize the behavior of your recipes, such as configuration settings or user names, without having to modify the cookbook.

You can, for example, use the same cookbook for different stacks and use custom JSON to specify key configuration settings for a particular stack. This saves you the time and effort required to modify the cookbook or use a different cookbook for each stack.

- You don't have to put potentially sensitive information such as database passwords in your cookbook repository.

You can instead use an attribute to define a default value and then use custom JSON to override that value with the real one.

For more information on how to use custom JSON to override attributes, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

The attribute file is named `default.rb` because it is a Ruby application, if a rather simple one. That means you can, for example, use conditional logic to specify attribute values based on the operating system. In [Conditional Logic \(p. 190\)](#), you specified a different subdirectory name for different Linux families in the recipe. With an attribute file, you can instead put the conditional logic in the attribute file.

The following attribute file uses `value_for_platform` to specify a different `['shared_dir']` attribute value, depending on the operating system. For other conditions, you can use Ruby `if-elsif-else` logic or a `case` statement.

```
data_dir = value_for_platform(
  "centos" => { "default" => "/srv/www/shared" },
  "ubuntu" => { "default" => "/srv/www/data" },
  "default" => "/srv/www/user_data"
)
default['createdir']['shared_dir'] = data_dir
default['createdir']['config_dir'] = "config"
default['createdir']['mode'] = 0755
default['createdir']['owner'] = 'root'
default['createdir']['group'] = 'root'
```

## To run the recipe

1. Run `kitchen destroy` to start with a fresh instance.
2. Replace the code in `attributes/default.rb` with the preceding example.

3. Edit `.kitchen.yml` to add a CentOS platform to the platforms section, as described in [Conditional Logic \(p. 190\)](#).
4. Run `kitchen converge`, and then log in to the instances to verify that the directories are there.

When you are finished, run `kitchen destroy` to terminate the instance. The next example uses a new cookbook.

## Example 6: Creating Files

After you have created directories, you often need to populate them with configuration files, data files, and so on. This topic shows two ways to install files on an instance.

### Topics

- [Installing a File from a Cookbook \(p. 195\)](#)
- [Creating a File from a Template \(p. 196\)](#)

## Installing a File from a Cookbook

The simplest way to install a file on an instance is to use a `cookbook_file` resource, which copies a file from the cookbook to a specified location on the instance. This topic extends the recipe from [Example 3: Creating Directories \(p. 187\)](#) to add a data file to `/srv/www/shared` after the directory is created. For reference, here is the original recipe.

```
directory "/srv/www/shared" do
  mode 0755
  owner 'root'
  group 'root'
  recursive true
  action :create
end
```

### To set up the cookbook

1. Inside the `opsworks_cookbooks` directory, create a directory named `createfile` and navigate to it.
2. Add a `metadata.rb` file to `createfile` with the following content.

```
name "createfile"
version "0.1.0"
```

3. Initialize and configure Test Kitchen, as described in [Example 1: Installing Packages \(p. 184\)](#), and remove CentOS from the `platforms` list.
4. Add a `recipes` subdirectory to `createfile`.

The file to be installed contains the following JSON data.

```
{
  "my_name" : "myname",
```

```
    "your_name" : "yourname",
    "a_number" : 42,
    "a_boolean" : true
}
```

### To set up the data file

1. Add a `files` subdirectory to `createfile` and a default subdirectory to `files`. Any file that you install with `cookbook_file` must be in a subdirectory of `files`, such as `files/default` in this example.

#### Tip

If you want to specify different files for different systems, you can put each system-specific file in a subfolder named for the system, such as `files/ubuntu`. The `cookbook_file` resource copies the appropriate system-specific file, if it exists, and otherwise uses the default file. For more information, see [cookbook\\_file](#).

2. Create a file named `example_data.json` with the JSON from the preceding example and add it to `files/default`.

The following recipe copies `example_data.json` to a specified.

```
directory "/srv/www/shared" do
  mode 0755
  owner 'root'
  group 'root'
  recursive true
  action :create
end

cookbook_file "/srv/www/shared/example_data.json" do
  source "example_data.json"
  mode 0644
  action :create_if_missing
end
```

After the directory resource creates `/srv/www/shared`, the `cookbook_file` resource copies `example_data.json` to that directory and also sets the file's user, group, and mode.

#### Note

The `cookbook_file` resource introduces a new action: `create_if_missing`. You could also use a `create` action, but that overwrites an existing file. If you don't want to overwrite anything, use `create_if_missing`, which installs `example_data.json` only if it does not already exist.

### To run the recipe

1. Run `kitchen destroy` to start with a fresh instance.
2. Create a `default.rb` file that contains the preceding recipe and save it to `recipes`.
3. Run `kitchen converge`, then log in to the instance to verify that `/srv/www/shared` contains `example_data.json`.

## Creating a File from a Template

The `cookbook_file` resource is useful for some purposes, but it just installs whatever file you have in the cookbook. A `template` resource provides a more flexible way to install files on an instance by creating

them dynamically from a template. You can then determine the details of the file's contents at runtime and change them as needed. For example, you might want a configuration file to have a particular setting when you start the instance and modify the setting later when you add more instances to the stack.

This example modifies the `createfile` cookbook to use a `template` resource to install a slightly modified version of `example_data.json`.

Here's what the installed file will look like.

```
{  
  "my_name" : "myname",  
  "your_name" : "yourname",  
  "a_number" : 42,  
  "a_boolean" : true,  
  "a_string" : "some string",  
  "platform" : "ubuntu"  
}
```

Template resources are typically used in conjunction with attribute files, so the example uses one to define the following values.

```
default['createfile']['my_name'] = 'myname'  
default['createfile']['your_name'] = 'yourname'  
default['createfile']['install_file'] = true
```

### To set up the cookbook

1. Delete the `createfile` cookbook's `files` directory and its contents.
2. Add an `attributes` subdirectory to `createfile` and add a `default.rb` file to `attributes` that contains the preceding attribute definitions.

A template is a `.erb` file that is basically a copy of the final file, with some of the contents represented by placeholders. When the `template` resource creates the file, it copies the template's contents to the specified file, and overwrites the placeholders with their assigned values. Here's the template for `example_data.json`.

```
{  
  "my_name" : "<%= node['createfile']['my_name'] %>",  
  "your_name" : "<%= node['createfile']['your_name'] %>",  
  "a_number" : 42,  
  "a_boolean" : <%= @a_boolean_var %>,  
  "a_string" : "<%= @a_string_var %>",  
  "platform" : "<%= node['platform'] %>"  
}
```

The `<%= . . . %>` values are the placeholders.

- `<%=node[ . . . ]%>` represents a node attribute value.

For this example, the `"your_name"` value is a placeholder that represents one of the attribute values from the cookbook's attribute file.

- <%=@...%> represents the value of a variable that is defined in the template resource, as discussed shortly.

## To create the template file

1. Add a `templates` subdirectory to the `createfile` cookbook and a `default` subdirectory to `templates`.

### Note

The `templates` directory works much like the `files` directory. You can put system-specific templates in a subdirectory such as `ubuntu` that is named for the system. The `template` resource uses the appropriate system-specific template if it exists and otherwise uses the `default` template.

2. Create a file named `example_data.json.erb` and put in the `templates/default` directory. The template name is arbitrary, but you usually create it by appending `.erb` to the file name, including any extensions.

The following recipe uses a `template` resource to create `/srv/www/shared/example_data.json`.

```
directory "/srv/www/shared" do
  mode 0755
  owner 'root'
  group 'root'
  recursive true
  action :create
end

template "/srv/www/shared/example_data.json" do
  source "example_data.json.erb"
  mode 0644
  variables(
    :a_boolean_var => true,
    :a_string_var => "some string"
  )
  only_if {node['createfile']['install_file']}
end
```

The `template` resource creates `example_data.json` from a template and installs it in `/srv/www/shared`.

- The template name, `/srv/www/shared/example_data.json`, specifies the installed file's path and name.
- The `source` attribute specifies the template used to create the file.
- The `mode` attribute specifies the installed file's mode.
- The resource defines two variables, `a_boolean_var` and `a_string_var`.

When the resource creates `example_data.json`, it overwrites the variable placeholders in the template with the corresponding values from the resource.

- The `only_if` guard attribute directs the resource to create the file only if `['createfile']['install_file']` is set to `true`.

### To run the recipe

1. Run `kitchen destroy` to start with a fresh instance.
2. Replace the code in `recipes/default.rb` with the preceding example.
3. Run `kitchen converge`, then log in to the instance to verify that the file is in `/srv/www/shared` and has the correct content.

When you are finished, run `kitchen destroy` to shut down the instance. The next section uses a new cookbook.

## Example 7: Running Commands and Scripts

Chef resources can handle a wide variety of tasks on an instance, but it is sometimes preferable to use a shell command or a script. For example, you might already have scripts that you use to accomplish certain tasks, and it will be easier to continue using them rather than implement new code. This section shows how to run commands or scripts on an instance.

### Topics

- [Running Commands \(p. 199\)](#)
- [Running Scripts \(p. 200\)](#)

## Running Commands

The `script` resource runs one or more commands. It supports the csh, bash, Perl, Python, and Ruby command interpreters. This topic shows how to run a simple bash command.

### To get started

1. Inside the `opsworks_cookbooks` directory, create a directory named `script` and navigate to it.
2. Add a `metadata.rb` file to `script` with the following content.

```
name "script"
version "0.1.0"
```

3. Initialize and configure Test Kitchen, as described in [Example 1: Installing Packages \(p. 184\)](#), and remove CentOS from the `platforms` list.
4. Inside `script`, create a directory named `recipes`.

You can run commands by using the `script` resource itself, but Chef also supports a set of command interpreter-specific versions of the resource, which are named for the interpreter. The following recipe uses a `bash` resource to run a simple bash script.

```
bash "install_something" do
  user "root"
  cwd "/tmp"
  code <<-EOH
    touch somefile
  EOH
  not_if do
    File.exists?("/tmp/somefile")
```

```
end  
end
```

The bash resource is configured as follows.

- It uses the default action, `run`, which runs the commands in the `code` block.

This example has one command, `touch somefile`, but a `code` block can contain multiple commands.

- The `user` attribute specifies the user that executes the command.
- The `cwd` attribute specifies the working directory.

For this example, `touch` creates a file in the `/tmp` directory.

- The `not_if` guard attribute directs the resource to take no action if the file already exists.

### To run the recipe

1. Create a `default.rb` file that contains the preceding example code and save it to `recipes`.
2. Run `kitchen converge`, then log in to the instance to verify that the file is in `/tmp`.

## Running Scripts

The `script` resource is convenient, especially if you need to run only one or two commands, but it's often preferable to store the script in a file and execute the file. The `execute` resource runs a specified executable file, including script files. This topic modifies the `script` cookbook from the preceding example to use `execute` to run a simple script. You can easily extend the example to more complex scripts, or other types of executable file.

### To set up the script file

1. Add a `files` subdirectory to `script` and a `default` subdirectory to `files`.
2. Create a file named `touchfile` that contains the following and add it to `files/default`.

```
touch somefile
```

This example use a single `touch` command, but the file can contain any number of commands.

The following recipe executes the script.

```
cookbook_file "/tmp/touchfile" do
  source "touchfile"
  mode 0755
end

execute "touchfile" do
  user "root"
  cwd "/tmp"
  command "./touchfile"
end
```

The `cookbook_file` resource copies the script file to `/tmp` and sets the mode to make the file executable. The `execute` resource then executes the file as follows:

- The `user` attribute specifies the command's user (`root` in this example).
- The `cwd` attribute specifies the working directory (`/tmp` in this example).
- The `command` attribute specifies the script to be executed (`touchfile` in this example), which is located in the working directory.

### To run the recipe

1. Replace the code in `recipes/default.rb` with the preceding example.
2. Run `kitchen converge`, then log in to the instance to verify that `/tmp` now contains the script file, with the mode set to 0755, and `somefile`.

When you are finished, run `kitchen destroy` to shut down the instance. The next section uses a new cookbook.

## Example 8: Managing Services

Packages such as application servers typically have an associated service that must be started, stopped, restarted, and so on. For example, you need to start the Tomcat service after installing the package or after the instance finishes booting, and restart the service each time you modify the configuration file. This topic discusses the basics of how to manage a service, using a Tomcat application server as an example.

### Note

The example does a very minimal Tomcat installation, just enough to demonstrate the basics of how to use a `service` resource. For an example of how to implement recipes for a more functional Tomcat server, see [Creating a Custom Tomcat Server Layer \(p. 240\)](#).

### Topics

- [Defining and Starting a Service \(p. 201\)](#)
- [Using notifies to Start or Restart a Service \(p. 203\)](#)

## Defining and Starting a Service

This section shows the basics of how to define and start a service.

### To get started

1. In the `opsworks_cookbooks` directory, create a directory named `tomcat` and navigate to it.
2. Add a `metadata.rb` file to `tomcat` with the following content.

```
name "tomcat"
version "0.1.0"
```

3. Initialize and configure Test Kitchen, as described in [Example 1: Installing Packages \(p. 184\)](#), and remove CentOS from the `platforms` list.
4. Add a `recipes` subdirectory to `tomcat`.

You use a `service` resource to manage a service. The following default recipe installs Tomcat and starts the service.

```
execute "install_updates" do
  command "apt-get update"
end

package "tomcat7" do
  action :install
end

include_recipe 'tomcat::service'

service 'tomcat' do
  action :start
end
```

The recipe does the following:

- The `execute` resource runs `apt-get update` to install the current system updates.

For the Ubuntu 12.04 LTS instance used in this example, you must install the updates before installing Tomcat. Other systems might have different requirements.

- The `package` resource installs Tomcat 7.
- The included `tomcat::service` recipe defines the service and is discussed later.
- The `service` resource starts the Tomcat service.

You can also use this resource to issue other commands, such as stopping and restarting the service.

The following example shows the `tomcat::service` recipe.

```
service 'tomcat' do
  service_name "tomcat7"
  supports :restart => true, :reload => false, :status => true
  action :nothing
end
```

This recipe creates the Tomcat service definition as follows:

- The resource name, `tomcat`, is used by other recipes to reference the service.

For example, `default.rb` references `tomcat` to start the service.

- The `service_name` resource specifies the Linux service name.

When you list the services on the instance, the Tomcat service will be named `tomcat7`.

- `supports` specifies how Chef manages the service's `restart`, `reload`, and `status` commands.
  - `true` indicates that Chef can use the init script or other service provider to run the command.
  - `false` indicates that Chef must attempt to run the command by other means.

Notice that `action` is set to `:nothing`, which directs the resource to take no action. The `service` resource does support actions such as `start` and `restart`. However, this cookbook follows a standard practice of using a service definition that takes no action and starting or restarting the service elsewhere. Each recipe that starts or restarts a service must first define it, so the simplest approach is to put the service definition in a separate recipe and include it in other recipes as needed.

**Note**

For simplicity, the default recipe for this example uses a `service` resource to start the service after running the service definition. A production implementation typically starts or restarts a service by using `notifies`, as discussed later.

**To run the recipe**

1. Create a `default.rb` file that contains the default recipe example and save it to `recipes`.
2. Create a `service.rb` file that contains the service definition example and save it to `recipes`.
3. Run `kitchen converge`, then log in to the instance and run the following command to verify that the service is running.

```
sudo service tomcat7 status
```

**Note**

If you were running `service.rb` separately from `default.rb`, you would have to edit `.kitchen.yml` to add `tomcat::service` to the run list. However, when you include a recipe, its code is incorporated into the parent recipe before the recipe is executed. `service.rb` is therefore basically a part of `default.rb` and doesn't require a separate run list entry.

## Using notifies to Start or Restart a Service

Production implementations typically do not use `service` to start or restart a service. Instead, they add `notifies` to any of several resources. For example, if you want to restart the service after modifying a configuration file, you include `notifies` in the associated `template` resource. Using `notifies` has the following advantages over using a `service` resource to explicitly restart the service.

- The `notifies` element restarts the service only if the associated configuration file has changed, so there's no risk of causing an unnecessary service restart.
- Chef restarts the service at most once at the end of each run, regardless of how many `notifies` the run contains.

For example, Chef run might include multiple template resources, each of which modifies a different configuration file and requires a service restart if the file has changed. However, you typically want to restart the service only once, at the end of the Chef run. Otherwise, you might attempt to restart a service that is not yet fully operational from an earlier restart, which can lead to errors.

This example modifies `tomcat::default` to include a `template` resource that uses `notifies` to restart the service. A realistic example would use a template resource that creates a customized version of one of the Tomcat configuration files, but those are rather long and complex. For simplicity, the example just uses the template resource from [Creating a File from a Template \(p. 196\)](#). It doesn't have anything to do with Tomcat, but it provides a simple way to show how to use `notifies`. For an example of how to use templates to create Tomcat configuration files, see [Setup Recipes \(p. 242\)](#).

**To set up the cookbook**

1. Add a `templates` subdirectory to `tomcat` and a `default` subdirectory to `templates`.
2. Copy the `example_data.json.erb` template from the `createfile` cookbook to the `templates/default` directory.
3. Add an `attributes` subdirectory to `tomcat`.
4. Copy the `default.rb` attribute file from the `createfile` cookbook to the `attributes` directory.

The following recipe uses `notifies` to restart the Tomcat service.

```
execute "install_updates" do
  command "apt-get update"
end

package "tomcat7" do
  action :install
end

include_recipe 'tomcat::service'

service 'tomcat' do
  action :enable
end

directory "/srv/www/shared" do
  mode 0755
  owner 'root'
  group 'root'
  recursive true
  action :create
end

template "/srv/www/shared/example_data.json" do
  source "example_data.json.erb"
  mode 0644
  variables(
    :a_boolean_var => true,
    :a_string_var => "some string"
  )
  only_if {node['createfile']['install_file']}
  notifies :restart, resources(:service => 'tomcat')
end
```

The example merges the recipe from [Creating a File from a Template \(p. 196\)](#) into the recipe from the preceding section, with two significant changes:

- The `service` resource is still there, but it now serves a somewhat different purpose.

The `:enable` action enables the Tomcat service at boot.

- The `template` resource now includes `notifies`, which restarts the Tomcat service if `example_data.json` has changed.

This ensures that the service is started when Tomcat is first installed and restarted after every configuration change.

### To run the recipe

1. Run `kitchen destroy` to start with a clean instance.
2. Replace the code in `default.rb` with the preceding example.
3. Run `kitchen converge`, then log in to the instance and verify that the service is running.

#### Tip

If you want to restart a service but the recipe doesn't include a resource such as `template` that supports `notifies`, you can instead use a dummy `execute` resource. For example

```
execute 'trigger tomcat service restart' do
  command '/bin/true'
  notifies :restart, resources(:service => 'tomcat')
end
```

The `execute` resource must have a `command` attribute, even if you are using the resource only as a way to run `notifies`. This example gets around that requirement by running `/bin/true`, which is a shell command that simply returns a success code.

## Example 9: Using Amazon EC2 Instances

To this point, you've been running instances locally in VirtualBox. While this is quick and easy, you will eventually want to test your recipes on an Amazon EC2 instance. In particular, if you want to run recipes on Amazon Linux, it is available only on Amazon EC2. You can use a similar system such as CentOS for preliminary implementation and testing, but the only way to fully test your recipes on Amazon Linux is with an Amazon EC2 instance.

This topic shows how to run recipes on an Amazon EC2 instance. You will use Test Kitchen and Vagrant in much the same way as the preceding sections, with two differences:

- The driver is `kitchen-ec2` instead of Vagrant.
- The cookbook's `.kitchen.yml` file must be configured with the information required to launch the Amazon EC2 instance.

### Tip

An alternative approach is to use the `vagrant-aws` Vagrant plug-in. For more information, see [Vagrant AWS Provider](#).

You will need AWS credentials to create an Amazon EC2 instance. If you don't have an AWS account you can obtain one, as follows.

### To sign up for AWS

1. Go to <http://aws.amazon.com>, and then click **Sign Up**.
2. Follow the on-screen instructions.

Part of the sign-up procedure involves receiving a phone call and entering a PIN using the phone keypad.

You should then [create an IAM user](#) with permissions to access Amazon EC2 and save the user's access and secret keys to a secure location on your workstation. Test Kitchen will use those credentials to create the instance. The preferred way to provide credentials to Test Kitchen is to assign the keys to the following environment variables on your workstation.

- `AWS_ACCESS_KEY` – your user's access key, which will look something like AKIAIOSFODNN7EXAMPLE.
- `AWS_SECRET_KEY` – your user's secret key, which will look something like wJalrXUtn-FEMI/K7MDENG/bPxRfiCYEXAMPLEKEY.

This approach reduces the chances of accidentally compromising your account by, for example, uploading a project containing your credentials to a public repository. For more information, see [Best Practices for Managing AWS Access Keys](#).

## To set up the cookbook

1. To use the `kitchen-ec2` driver, you must have the `ruby-dev` package installed on your system. The following example command shows how to use `aptitude` to install the package on a Ubuntu system.

```
sudo aptitude install ruby1.9.1-dev
```

2. The `kitchen-ec2` driver is a gem, which you can install as follows:

```
gem install kitchen-ec2
```

Depending on your workstation, this command might require `sudo`, or you can also use a Ruby environment manager such as [RVM](#).

3. You must specify an Amazon EC2 SSH key pair that Test Kitchen can use to connect to the instance. If you don't have an Amazon EC2 key pair, see [Amazon EC2 Key Pairs](#) for information on how to create one. Note that the key pair must belong to the same AWS region as the instance. The example uses US East (Northern Virginia).

After you have selected a key pair, create a subdirectory of `opsworks_cookbooks` named `ec2_keys` and copy the key pair's private key (`.pem`) file to that subdirectory. Note that putting the private key in `ec2_keys` is just a convenience that simplifies the code a bit; it can be anywhere on your system.

4. Create a subdirectory of `opsworks_cookbooks` named `createdir-ec2` and navigate to it.
5. Add a `metadata.rb` file to `createdir-ec2` with the following content.

```
name "createdir-ec2"
version "0.1.0"
```

6. Initialize Test Kitchen, as described in [Example 1: Installing Packages \(p. 184\)](#). The following section describes how to configure `.kitchen.yml`, which is significantly more complicated for Amazon EC2 instances.
7. Add a `recipes` subdirectory to `createdir-ec2`.

## Configuring `.kitchen.yml` for Amazon EC2

You configure `.kitchen.yml` with the information that the `kitchen-ec2` driver needs to launch an appropriately configured Amazon EC2 instance. The following is an example of a `.kitchen.yml` file for an Amazon Linux instance in the US East (Northern Virginia) region.

```
---
driver:
  name: ec2
  aws_ssh_key_id: US-East1
  ssh_key: ../ec2_keys/US-East1.pem
  region: us-east-1
  availability_zone: us-east-1c
  require_chef_omnibus: true

provisioner:
  name: chef_solo
```

```
platforms:  
  - name: amazon  
    driver:  
      image_id: ami-ed8e9284  
      username: ec2-user  
  
suites:  
  - name: default  
    run_list:  
      - recipe[createdir-ec2::default]  
    attributes:
```

You can use the default settings for the `provisioner` and `suites` sections, but you must modify the default `driver` and `platforms` settings. This example uses a minimal list of settings, and accepts the default values for the remainder. For a complete list of `kitchen-ec2` settings, see [Kitchen::Ec2: A Test Kitchen Driver for Amazon EC2](#).

The example sets the following `driver` attributes. It assumes that you have assigned your user's access and secret keys to the standard environment variables, as discussed earlier. The driver uses those keys by default. Otherwise, you must explicitly specify the keys by adding `aws_access_key_id` and `aws_secret_access_key` to the `driver` attributes, set to the appropriate key values.

**name**

(Required) This attribute must be set to `ec2`.

**aws\_ssh\_key\_id**

(Required) The Amazon EC2 SSH key pair name, which is named `US-East1` in this example.

**ssh\_key**

(Required) The private key (`.pem`) file for the key that you specified for `aws_ssh_key_id`. For this example, the file is named `US-East1.pem` and is in the `../opsworks/ec2_keys` directory.

**region**

(Required) The instance's AWS region. The example uses US East (Northern Virginia), which is represented by `us-east-1`.

**availability\_zone**

(Optional) The instance's Availability Zone. If you omit this setting, Test Kitchen uses a default Availability Zone for the specified region, which is `us-east-1b` for US East (Northern Virginia). However, the default zone might not be available for your account. In that case, you must explicitly specify an Availability Zone. As it happens, the account used to prepare the examples doesn't support `us-east-1b`, so the example explicitly specifies `us-east-1c`.

**security\_group\_ids**

(Optional) A list of security group IDs to apply to the instance. The example omits this setting, which applies the default security group to the instance. Make sure that the security group ingress rules allow inbound SSH connections, or Test Kitchen will not be able to communicate with the instance. If you use the default security group, you might need to edit it accordingly. For more information, see [Amazon EC2 Security Groups](#).

**Important**

Rather than use your account credentials for the access and secret keys, you should create an IAM user and provide those credentials to Test Kitchen. For more information, see [Best Practices for Managing AWS Access Keys](#).

Be careful not to put `.kitchen.yml` in a publicly accessible location, such as uploading it to a public GitHub or Bitbucket repository. Doing so exposes your credentials and could compromise your account's security.

The `kitchen-ec2` driver provides default support for the following platforms:

- ubuntu-10.04
- ubuntu-12.04
- ubuntu-12.10
- ubuntu-13.04
- centos-6.4
- debian-7.1.0

If you want to use one or more of these platforms, add the appropriate platform names to `platforms`. The `kitchen-ec2` driver automatically selects an appropriate AMI and generates an SSH user name. You can use other platforms—this example uses Amazon Linux—but you must explicitly specify the following `platforms` attributes.

**name**

The platform name. This example uses Amazon Linux, so `name` is set to `amazon`.

**driver**

The `driver` attributes, which include the following:

- `image_id` – The platform's AMI, which must belong to the specified region. The example uses `ami-ed8e9284`, an Amazon Linux AMI from the US East (Northern Virginia) region.
- `username` – The SSH user name that Test Kitchen will use to communicate with the instance.

Use `ec2-user` for Amazon Linux. Other AMIs might have different user names.

Replace the code in `.kitchen.yml` with the example, and assign appropriate values to account-specific attributes such as `aws_access_key_id`.

## Running the Recipe

This example uses the recipe from [Iteration \(p. 189\)](#).

### To run the recipe

1. Create a file named `default.rb` with the following code and save it to the cookbook's `recipes` folder.

```
directory "/srv/www/shared" do
  mode 0755
  owner 'root'
  group 'root'
  recursive true
  action :create
end
```

2. Run `kitchen converge` to execute the recipe. Note that this command will take longer to complete than the previous examples because of the time required to launch and initialize an Amazon EC2 instance.
3. Go to the [Amazon EC2 console](#), select the US East (Northern Virginia) region, and click **Instances** in the navigation pane. You will see the newly created instance in the list.
4. Run `kitchen login` to log in to the instance, just as you have been doing for instances running in VirtualBox. You will see the newly created directories under `/srv`. You can also use your favorite SSH client to connect to the instance.

## Next Steps

This chapter walked you through the basics of how to implement Chef cookbooks, but there's much more:

- The examples showed you how to use some of the more commonly used resources, but there are many more.

For the resources that were covered, the examples used only some of the available attributes and actions. For a complete reference, see [About Resources and Providers](#).

- The examples used only the core cookbook elements: `recipes`, `attributes`, `files`, and `templates`.

Cookbooks can also include a variety of other elements, such as `libraries`, `definitions`, and `specs`. For more information, see the [Chef documentation](#).

- The examples used Test Kitchen only as a convenient way to start instances, run recipes, and log in to instances.

Test Kitchen is primarily a testing platform that you can use to run a variety of tests on your recipes. If you haven't done so already, go through the rest of the [Test Kitchen walkthrough](#), which introduces you to its testing features.

## Implementing Cookbooks for AWS OpsWorks

[Cookbook Basics \(p. 181\)](#) introduced you to cookbooks and recipes. The examples in that section were simple by design and will work on any instance that supports Chef, including AWS OpsWorks instances. To implement more sophisticated cookbooks for AWS OpsWorks, you typically need to take full advantage of the AWS OpsWorks environment, which differs from standard Chef in a number of ways.

This topic describes the basics of implementing recipes for AWS OpsWorks instances.

**Tip**

If you are not familiar with how to implement cookbooks, you should start with [Cookbook Basics \(p. 181\)](#).

**Topics**

- [Running a Recipe on an AWS OpsWorks Instance \(p. 209\)](#)
- [Mocking the Stack Configuration and Deployment JSON on Vagrant \(p. 213\)](#)

## Running a Recipe on an AWS OpsWorks Instance

Test Kitchen and Vagrant provide a simple and efficient way to implement cookbooks, but to verify that a cookbook's recipes will run correctly in production, you must run them on an AWS OpsWorks instance. This topic describes how to install a custom cookbook on an AWS OpsWorks instance and run a simple recipe. It also provides some tips for efficiently fixing recipe bugs.

First, you need to create a stack. The following briefly summarizes how to create a stack for this example. For more information, see [Create a New Stack \(p. 41\)](#).

1. Open the [AWS OpsWorks console](#) and click **Add Stack**.
2. Specify the following settings, accept the defaults for the other settings, and click **Add Stack**.
  - **Name** – OpsTest
  - **Default SSH key** – An Amazon EC2 key pair

If you need to create an Amazon EC2 key pair, see [Amazon EC2 Key Pairs](#). Note that the key pair must belong to the same AWS region as the instance. The example uses the default US East (Northern Virginia) region.

3. Click **Add a layer** and [add a custom layer \(p. 96\)](#) to the stack with the following settings.
  - **Name** – OpsTest
  - **Short name** – opstest

Any layer type will actually work, but the example doesn't require any of the packages that are installed by the other layer types, so a custom layer is the simplest approach.

4. [Add a 24/7 instance \(p. 103\)](#) with default settings to the layer and [start it \(p. 109\)](#).

While the instance is starting up—it usually takes several minutes—you can create the cookbook. This example will use a slightly modified version of the recipe from [Conditional Logic \(p. 190\)](#), which creates a data directory whose name depends on the platform.

### To set up the cookbook

1. Create a directory within `opsworks_cookbooks` named `opstest` and navigate to it.
2. Create a `metadata.rb` file with the following content and save it to `opstest`.

```
name "opstest"
version "0.1.0"
```

3. Create a `recipes` directory within `opstest`.
4. Create a `default.rb` file with the following recipe and save it to the `recipes` directory.

```
Chef::Log.info("*****Creating a data directory.*****")

data_dir = value_for_platform(
  "centos" => { "default" => "/srv/www/shared" },
  "ubuntu" => { "default" => "/srv/www/data" },
  "default" => "/srv/www/config"
)

directory data_dir do
  mode 0755
  owner 'root'
  group 'root'
  recursive true
  action :create
end
```

Notice that the recipe logs a message, but it does so by calling `Chef::Log.info`. You aren't using Test Kitchen for this example, so the `log` method isn't very useful. `Chef::Log.info` puts the message into the Chef log, which you can read after the Chef run is finished. AWS OpsWorks provides an easy way to view these logs, as described later.

#### Tip

Chef logs usually contain a lot of routine and relatively uninteresting information. The '\*' characters bracketing the message text make it easier to spot.

5. Create a `.zip` archive of `opsworks_cookbooks`. To install your cookbook on an AWS OpsWorks instance, you must store it in a repository and provide AWS OpsWorks with the information required to download the cookbook to the instance. You can store your cookbooks in any of several supported repository types. This example stores an archive file containing the cookbooks in an Amazon S3 bucket. For more information on cookbook repositories, see [Cookbook Repositories \(p. 154\)](#).

**Tip**

For simplicity, this example just archives the entire `opsworks_cookbooks` directory. However, it means that AWS OpsWorks will download all the cookbooks in `opsworks_cookbooks` to the instance, even though you will use only one of them. To install only the example cookbook, create another parent directory and move `opstest` to that directory. Then create a `.zip` archive of the parent directory and use it instead of `opsworks_cookbooks.zip`.

6. [Upload the archive to an Amazon S3 bucket, make the archive public](#), and record the archive's URL. It should look something like `https://s3.amazonaws.com/cookbook_bucket/opsworks_cookbooks.zip`.

**Tip**

This approach is simple, but it allows anyone to download the cookbooks, which is usually not acceptable for production code. To limit access, leave the archive private and specify an instance profile for the stack that grants Amazon S3 permissions to applications running on the stack's instances.

You can now install the cookbook and run the recipe.

### To run the recipe

1. [Edit the stack to enable custom cookbooks \(p. 171\)](#), and specify the following settings.
  - **Repository type – Http Archive**
  - **Repository URL** – The cookbook archive URL that you recorded earlierUse the default values for the other settings and click **Save** to update the stack configuration.
2. [Run the Update Custom Cookbooks stack command \(p. 52\)](#), which installs the current version of your custom cookbooks on the stack's instances. If an earlier version of your cookbooks is present, this command overwrites it.
3. Execute the recipe by running the **Execute Recipes** stack command with **Recipes to execute** set to `opstest::default`. This command initiates a Chef run, with a run list that consists of `opstest::default`.

**Tip**

You typically have AWS OpsWorks [run your recipes automatically \(p. 177\)](#) by assigning them to the appropriate lifecycle event. You can run such recipes by manually triggering the event. You can use a stack command to trigger Setup and Configure event, and a [deploy command \(p. 132\)](#) to trigger Deploy and Undeploy.

After the recipe runs successfully, you can verify it.

### To verify opstest

1. The first step is to examine the [Chef log \(p. 338\)](#). Click **show** in the `opstest1` instance's **Log** column to display the log. Scroll down and you will see your log message near the bottom.

...

```
[2014-07-31T17:01:45+00:00] INFO: Storing updated cook
books/opsworks_cleanup/attributes/customize.rb in the cache.
[2014-07-31T17:01:45+00:00] INFO: Storing updated cook
books/opsworks_cleanup/metadata.rb in the cache.
[2014-07-31T17:01:46+00:00] INFO: *****Creating a data directory.*****
[2014-07-31T17:01:46+00:00] INFO: Processing template[/etc/hosts] action
create (opsworks_stack_state_sync::hosts line 3)
...
```

2. Use SSH to log in to the instance (p. 119) and list the contents of /srv/www/.

If you followed all the steps, you will see /srv/www/config rather than the /srv/www/shared directory you were expecting. The following section provides some guidelines for quickly fixing such bugs.

## Troubleshooting and Fixing Recipes

If you aren't getting the expected results, or your recipes don't even run successfully, you need to troubleshoot the problems and fix any bugs. Troubleshooting typically starts by examining the Chef log. It contains a detailed description of the run and includes any inline log messages from your recipes. The logs are particularly useful if your recipe simply failed. When that happens, Chef logs the error, including a stack trace.

If the recipe was successful, as it was for this example, the Chef log often isn't much help. In this case, you can figure out the problem by just taking a closer look at the recipe, the first few lines in particular:

```
Chef::Log.info("*****Creating a data directory.*****")

data_dir = value_for_platform(
  "centos" => { "default" => "/srv/www/shared" },
  "ubuntu" => { "default" => "/srv/www/data" },
  "default" => "/srv/www/config"
)
...
```

CentOS is a reasonable stand-in for Amazon Linux when you are testing recipes on Vagrant, but now you are running on an actual Amazon Linux instance. The platform value for Amazon Linux is `amazon`, which isn't included in the `value_for_platform` call, so the recipe creates /srv/www/config by default. For more information on troubleshooting, see [Debugging and Troubleshooting Guide \(p. 337\)](#).

Now that you have identified the problem, you need to update the recipe and verify the fix. You could go back to the original source files, update `default.rb`, upload a new archive to Amazon S3, and so on. However, that process can be a bit tedious and time consuming. The following shows a much quicker approach that is especially useful for simple recipe bugs like the one in the example: edit the recipe on the instance.

### To edit a recipe on an instance

1. Use SSH to log in to the instance and then run `sudo su` to elevate your privileges. You need root privileges to access the cookbook directories.
2. AWS OpsWorks stores your cookbook in `/opt/aws/opsworks/current/site-cookbooks`, so navigate to `/opt/aws/opsworks/current/site-cookbooks/opstest/recipes`.

#### Tip

AWS OpsWorks also stores a copy of your cookbooks in `/opt/aws/opsworks/current/merged-cookbooks`. Don't edit that cookbook. When you execute the recipe, AWS

OpsWorks copies the cookbook from `.../site-cookbooks` to `.../merged-cookbooks`, so any changes you make in `.../merged-cookbooks` will be overwritten.

3. Use a text editor on the instance to edit `default.rb`, and replace `centos` with `amazon`. Your recipe should now look like the following.

```
Chef::Log.info("*****Creating a data directory.*****")  
  
data_dir = value_for_platform(  
  "amazon" => { "default" => "/srv/www/shared" },  
  "ubuntu" => { "default" => "/srv/www/data" },  
  "default" => "/srv/www/config"  
)  
...  
...
```

To verify the fix, execute the recipe by running the **Execute Recipe** stack command again. The instance should now have a `/srv/www/shared` directory. If you need to make further changes to a recipe, you can run **Execute Recipe** as often as you like; you don't need to stop and restart the instance each time you run the command. When you are satisfied that the recipe is working correctly, don't forget to update the code in your source cookbook.

**Tip**

If you have assigned your recipe to a lifecycle event so AWS OpsWorks runs it automatically, you can always use **Execute Recipe** to rerun the recipe. You can also rerun the recipe as many times as you want without restarting the instance by using the AWS OpsWorks console to manually trigger the appropriate event. This approach runs all of the event's recipes. Here's a reminder:

- Use a stack command to trigger Setup or Configure events.
- Use a deploy command to trigger Deploy or Undeploy events.

## Mocking the Stack Configuration and Deployment JSON on Vagrant

AWS OpsWorks installs a [stack configuration and deployment JSON \(p. 262\)](#) on each instance in your stack for every lifecycle event. The attributes in this JSON provide a snapshot of the stack configuration, including the configuration of each layer and its online instances, the configuration of each deployed app, and so on. AWS OpsWorks adds the stack configuration and deployment JSON attributes to the node object, so they can be accessed by any recipe using Chef node syntax; most recipes for AWS OpsWorks instances use one or more of these attributes.

An instance running in a Vagrant box is not managed by AWS OpsWorks, so it does not have a stack configuration and deployment JSON. However, you can mock stack configuration and deployment JSON on the instance by adding attribute definitions from a typical JSON to your Test Kitchen environment. Test Kitchen then adds the attributes to the instance's node object, and your recipes can access the attributes using the same node syntax that they would on an AWS OpsWorks instance.

This topic shows how to obtain a copy of a suitable stack configuration and deployment JSON, install it on an instance, and access the attributes.

**Tip**

If you are using Test Kitchen to run tests on your recipes, [fauxhai](#) provides an alternative way to mock stack configuration and deployment JSON.

## To set up the cookbook

1. Create a subdirectory of `opsworks_cookbooks` named `printjson` and navigate to it.
2. Initialize and configure Test Kitchen, as described in [Example 1: Installing Packages \(p. 184\)](#).
3. Add two subdirectories to `printjson`: `recipes` and `environments`.

You could mock stack configuration and deployment JSON by adding an attribute file to your cookbook with the appropriate definitions, but a better approach is to use the Test Kitchen environment. There are two basic approaches:

- Add attribute definitions to `.kitchen.yml`.

This approach is most useful if you have just a few attributes. For more information, see [kitchen.yml](#).

- Define the attributes in an environment file and reference the file in `.kitchen.yml`.

This approach is usually preferable for stack configuration and deployment JSON because the environment file is already in JSON format; you can just paste it in. All of the examples use an environment file.

The simplest way to create a stack configuration and deployment JSON for your cookbook is to create an appropriately configured stack and copy the resulting JSON. To keep your Test Kitchen environment file manageable, you can then edit that JSON to have only the attributes that your recipes need. The examples in this chapter are based on the stack from [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#), which is a simple PHP application server stack with a load balancer, PHP application servers, and a MySQL database server.

## To create a stack configuration and deployment JSON

1. Create MyStack as described in [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#), including deploying SimplePHPApp. If you prefer, you can omit the second PHP App Server instance called for in [Step 4: Scale Out MyStack \(p. 31\)](#); the examples don't use those attributes.
2. If you haven't already done so, start the `php-app1` instance, and then [log in with SSH \(p. 119\)](#).
3. In the terminal window, run the following [agent cli \(p. 368\)](#) command:

```
sudo opsworks-agent-cli get_json
```

This command prints the instance's most recent stack configuration and deployment JSON to the terminal window.

4. Copy the JSON to a `.json` file and save it in a convenient location on your workstation. The details depend on your SSH client. For example, if you are using PUTTY on Windows, you can run the `Copy All to Clipboard` command, which copies all the text in the terminal window to the Windows clipboard. You can then paste the contents into a `.json` file and edit the file to remove any extraneous text.
5. Create edited versions of the MyStack JSON as needed. Stack configuration and deployment JSON has a large number of attributes and cookbooks typically use only a small subset of them. To keep your environment file manageable, you can edit the stack configuration and deployment JSON so that it contains only the attributes that your cookbooks actually use.

This example uses a heavily edited version of the MyStack JSON that includes just two `['opsworks'][['stack']]` attributes, `['id']` and `['name']`. Create an edited version of the MyStack JSON that looks something like the following:

```
{  
  "opsworks": {  
    "stack": {  
      "name": "MyStack",  
      "id": "42dfd151-6766-4f1c-9940-ba79e5220b58",  
    },  
  },  
}
```

To get this JSON into the instance's node object, you need to add it to a Test Kitchen environment.

### To add stack configuration and deployment JSON to the Test Kitchen environment

1. Create an environment file named `test.json` with the following contents and save it to the cookbook's `environments` folder.

```
{  
  "default_attributes": {  
    "opsworks" : {  
      "stack" : {  
        "name" : "MyStack",  
        "id" : "42dfd151-6766-4f1c-9940-ba79e5220b58"  
      }  
    }  
  },  
  "chef_type" : "environment",  
  "json_class" : "Chef::Environment"  
}
```

The environment file has the following elements:

- `default_attributes` – The default attributes in JSON format.

These attributes are added to the node object with the `default` attribute type, which is the type used by all of the stack configuration and deployment JSON attributes. This example uses the edited version of the stack configuration and deployment JSON shown earlier.

- `chef_type` – Set this element to `environment`.
- `json_class` – Set this element to `Chef::Environment`.

2. Edit `.kitchen.yml` to define the Test Kitchen environment, as follows.

```
---  
driver:  
  name: vagrant  
  
provisioner:  
  name: chef_solo  
  environments_path: ./environments  
  
platforms:
```

```
- name: ubuntu-12.04

suites:
  - name: printjson
    provisioner:
      solo_rb:
        environment: test
    run_list:
      - recipe[printjson::default]
    attributes:
```

You define the environment by adding the following elements to the default .kitchen.yml created by kitchen init.

#### **provisioner**

Add the following elements.

- **name** – Set this element to `chef_solo`.

To replicate the AWS OpsWorks environment more closely, you could use [Chef client local mode](#) instead of Chef solo. Local mode is a Chef client option that uses a lightweight version of Chef Server (Chef Zero) that runs locally on the instance instead of a remote server. It enables your recipes to use Chef Server features such as search or data bags without connecting to a remote server.

- **environments\_path** – The cookbook subdirectory that contains the environment file, `./environments`, for this example.

#### **suites:provisioner**

Add a `solo_rb` element with an `environment` element set to the environment file's name, minus the `.json` extension. This example sets `environment` to `test`.

3. Create a recipe file named `default.rb` with the following content and save it to the cookbook's `recipes` directory.

```
log "Stack name: #{node['opsworks']['stack']['name']}"
log "Stack id: #{node['opsworks']['stack']['id']}
```

This recipe simply logs the two stack configuration and deployment JSON values that you added to the environment. Although the recipe is running locally in Virtual Box, you reference those attributes using the same node syntax that you would if the recipe were running on an AWS OpsWorks instance.

4. Run `kitchen converge`. You should see something like the following log output.

```
...
Converging 2 resources
Recipe: printjson::default
  * log[Stack name: MyStack] action write[2014-07-01T23:14:09+00:00] INFO:
    Processing log[Stack name: MyStack] action write (printjson::default line
    1)
[2014-07-01T23:14:09+00:00] INFO: Stack name: MyStack

  * log[Stack id: 42dfd151-6766-4f1c-9940-ba79e5220b58] action write[2014-
    07-01T23:14:09+00:00] INFO: Processing log[Stack id: 42dfd151-6766-4f1c-
    9940-ba79e5220b58] action write (printjson::default line 2)
[2014-07-01T23:14:09+00:00] INFO: Stack id: 42dfd151-6766-4f1c-9940-
```

ba79e5220b58

...

# Resource Management

---

The **Resources** page enables you to use any of your account's [Elastic IP address](#), [Amazon EBS volume](#), or [Amazon RDS instance](#) resources in an AWS OpsWorks stack. You can use **Resources** to do the following:

- [Register a resource \(p. 219\)](#) with a stack, which allows you to attach the resource to one of the stack's instances.
- [Attach a resource \(p. 222\)](#) to one of the stack's instances.
- [Move a resource \(p. 222\)](#) from one instance to another.
- [Detach a resource \(p. 226\)](#) from an instance. The resource remains registered and can be attached to another instance.
- [Deregister a resource \(p. 228\)](#). An unregistered resource cannot be used by AWS OpsWorks, but it remains in your account unless you delete it, and can be registered with another stack.

Note the following constraints:

- The Resources page manages standard or PIOPS Amazon EBS volumes, but not RAID arrays.
- You can't attach an Amazon EBS volume to or detach it from a running instance.

You can operate only on offline instances. For example, you can register an in-use volume with a stack and attach it to an offline instance, but you must stop the original instance and detach the volume before starting the new instance. Otherwise, the start process will fail.

- You can attach an Elastic IP address to and detach it from a running instance.

You can operate on online or offline instances. For example, you can register an in-use address and assign it to a running instance, and AWS OpsWorks will automatically reassign the address.

- To register and deregister all resources, your IAM policy must grant permissions for the following actions:
  - [RegisterElasticIp](#)
  - [UpdateElasticIp](#)
  - [DeregisterElasticIp](#)
  - [RegisterVolume](#)
  - [UpdateVolume](#)
  - [DeregisterVolume](#)
  - [RegisterRdsDbInstance](#)
  - [UpdateRdsDbInstance](#)
  - [DeregisterRdsDbInstance](#)

The [Manage permissions level \(p. 282\)](#) grants permissions for all of these actions. To prevent a Manage user from registering or deregistering particular resources, edit their IAM policy to deny permissions for the appropriate actions. For more information, see [Security and Permissions \(p. 276\)](#).

The following sections describe how to use the **Resources** page to manage your Elastic IP address and Amazon EBS volume resources.

#### Topics

- [Registering Resources with a Stack \(p. 219\)](#)
- [Attaching and Moving Resources \(p. 222\)](#)
- [Detaching Resources \(p. 226\)](#)
- [Deregistering Resources \(p. 228\)](#)

## Registering Resources with a Stack

Amazon EBS volumes or Elastic IP addresses must be registered with a stack before you can attach them to instances. When AWS OpsWorks creates resources for a stack, they are automatically registered with that stack. If you want to use externally created resources, you must explicitly register them. Note the following:

- You can register a resource with only one stack at a time.
- When you delete a stack, AWS OpsWorks deregisters all resources.

#### Topics

- [Registering Amazon EBS Volumes with a Stack \(p. 219\)](#)
- [Registering Elastic IP Addresses with a Stack \(p. 220\)](#)
- [Registering Amazon RDS Instances with a Stack \(p. 221\)](#)

## Registering Amazon EBS Volumes with a Stack

Use the following procedure to register Amazon EBS volumes.

#### To register an Amazon EBS volume

1. Open the desired stack and click **Resources** in the navigation pane.
2. Click **Volumes** to display the available Amazon EBS volumes. Initially, the stack has no registered volumes, as shown in the following illustration.

Resources

Show Unregistered Volumes

Volumes    Elastic IPs    RDS    Search

No volumes have been registered yet. [Show unregistered volumes.](#)



3. Click **Show Unregistered Volumes** to display the Amazon EBS volumes in your account that are in the stack's region and if applicable, the stack's VPC. The **Status** column indicates whether the volumes are available for use. **Volume Type** indicates whether the volume is standard (standard) or PIOPS (io1, followed by the IOPS per disk value in parentheses).

## Resources Unregistered Volumes

The list contains only volumes created in **us-east-1**. Add a Volume on [EC2](#).

Name	EC2 ID	EC2 Instance ID	Size (GiB)	Device	Volume Type	AZ	Status
Disk 1 of 2	vol-3753f475		50		standard	us-east-1a	available
Disk 2 of 2	vol-eb54f3a9		50		standard	us-east-1a	available
PHP-LB-Standard	vol-6a4bec28		100		standard	us-east-1a	available
no name	vol-68702625	<a href="#">I-9a5328ba</a>	8	/dev/sda1	standard	us-east-1c	in-use

[Cancel](#) [Register with Stack](#)

4. Select the appropriate volumes and click **Register to Stack**. The **Resources** page now lists the newly registered volumes.

## Resources

[Show Unregistered Volumes](#)

Name	EC2 ID	Instance	Size (GiB)	Volume Type	AZ	Actions
PHP-LB-Standard	vol-6a4bec28	<a href="#">assign to instance</a>	100	standard	us-east-1a	<a href="#">edit</a>

[+ Unregistered Volumes](#)

To register additional volumes, click **Show Unregistered Volumes** or **+ Unregistered Volumes** and repeat this procedure.

## Registering Elastic IP Addresses with a Stack

Use the following procedure to register Elastic IP addresses.

### To register an Elastic IP address

1. Open the stack's **Resources** page and click **Elastic IPs** to display the available Elastic IP addresses. Initially, the stack has no registered addresses, as shown in the following illustration.

## Resources

[Show Unregistered Elastic IPs](#)

No Elastic IPs have been registered yet. [Show unregistered Elastic IPs](#).

2. Click **Show Unregistered Elastic IPs** to display the available Elastic IP addresses in your account that are in the stack's region.

## Resources Unregistered Elastic IPs

Address	Instance	Domain
192.0.2.0		standard
192.0.2.10		standard
192.0.2.20		standard

Cancel **Register with Stack**

3. Select the appropriate addresses and click **Register to Stack**. This returns you to the **Resources** page, which now lists the newly registered addresses.

## Resources

Address	Name	Instance	Public DNS	Actions
192.0.2.0	-	associate with instance	-	edit

Show Unregistered Elastic IPs **+ Unregistered Elastic IPs**

To register additional addresses, click **Show Unregistered Elastic IPs** or **+ Unregistered Elastic IPs** and repeat this procedure.

## Registering Amazon RDS Instances with a Stack

Use the following procedure to register Amazon RDS instances.

### To register an Amazon RDS instance

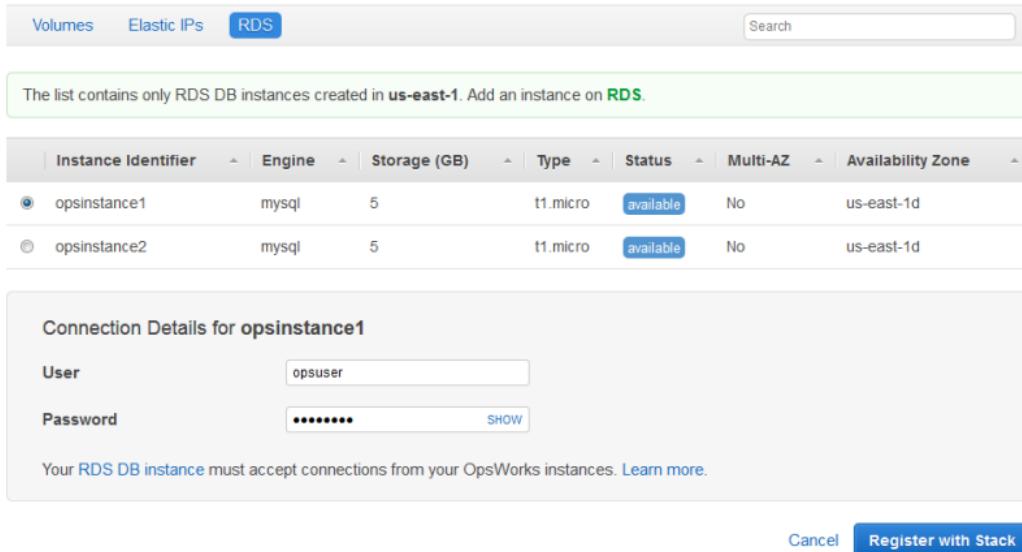
1. Open the stack's **Resources** page and click **RDS** to display the available Amazon RDS instances. Initially, the stack has no registered instances, as shown in the following illustration.

## Resources

No RDS DB instances have been registered yet. **Show unregistered RDS DB instances**.

2. Click **Show Unregistered RDS DB instances** to display the available Amazon RDS instances in your account that are in the stack's region.

## Resources Unregistered RDS DB instances



Instance Identifier	Engine	Storage (GB)	Type	Status	Multi-AZ	Availability Zone
opsinstance1	mysql	5	t1.micro	available	No	us-east-1d
opsinstance2	mysql	5	t1.micro	available	No	us-east-1d

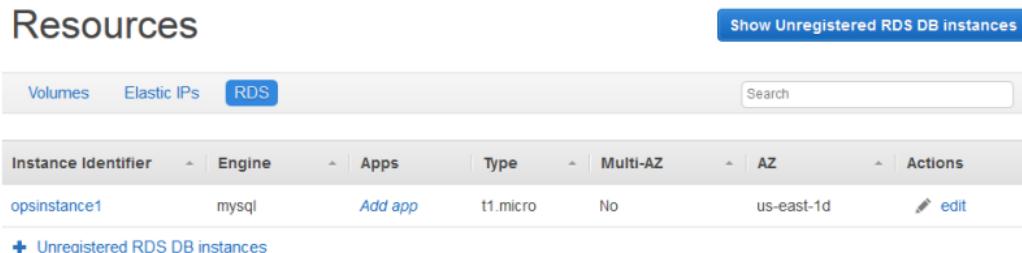
Connection Details for opsinstance1

User	<input type="text" value="opsuser"/>
Password	<input type="password" value="*****"/> <a href="#">SHOW</a>

Your RDS DB instance must accept connections from your OpsWorks instances. [Learn more.](#)

[Cancel](#) [Register with Stack](#)

3. Select the appropriate instance, enter its master user and master password values for **User** and **Password**, and click **Register to Stack**. This returns you to the **Resources** page, which now lists the newly registered instance.



Instance Identifier	Engine	Apps	Type	Multi-AZ	AZ	Actions
opsinstance1	mysql	<a href="#">Add app</a>	t1.micro	No	us-east-1d	<a href="#"> edit</a>

[+ Unregistered RDS DB instances](#)

### Important

You must ensure that the user and password that you use to register the Amazon RDS instance correspond to a valid user and password. If they do not, your applications will not be able connect to the instance.

To register additional addresses, click **Show Unregistered RDS DB instances** or **+ Unregistered RDS DB instances** and repeat this procedure. For more information about how to use Amazon RDS instances with AWS OpsWorks, see [Amazon RDS Service Layer \(p. 77\)](#).

### Note

You can also register Amazon RDS instances through the **Layers** page. For more information, see [Amazon RDS Service Layer \(p. 77\)](#).

## Attaching and Moving Resources

After you register a resource with a stack, you can attach it to one of the stack's instances. You can also move an attached resource from one instance to another. Note the following:

- When you attach or move Amazon EBS volumes, the instances involved in the operation must be offline. If the instance you are interested in is not on the **Resources** page, go to the **Instances** page and [stop](#)

the instance (p. 109). After it has stopped, you can return to the **Resources** page and attach or move the resource.

- When you attach or move Elastic IP addresses, the instances can be online or offline.
- If you delete an instance, any attached resources remain registered with the stack. You can then attach the resource to another instance or, if you no longer need it, deregister the resource.

#### Topics

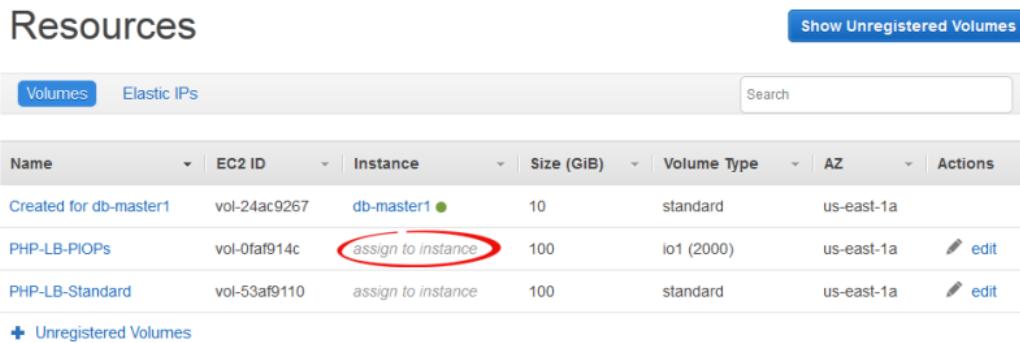
- [Assigning Amazon EBS Volumes to an Instance \(p. 223\)](#)
- [Associating Elastic IP Addresses with an Instance \(p. 224\)](#)
- [Attaching Amazon RDS Instances to an App \(p. 226\)](#)

## Assigning Amazon EBS Volumes to an Instance

You can assign a registered Amazon EBS volume to an instance and move it from one instance to another, but both instances must be offline.

### To assign an Amazon EBS volume to an instance

1. On the Resources page, click **assign to instance** in the appropriate volume's **Instance** column.



Name	EC2 ID	Instance	Size (GiB)	Volume Type	AZ	Actions
Created for db-master1	vol-24ac9267	db-master1 ●	10	standard	us-east-1a	
PHP-LB-PIOPS	vol-0faf914c	assign to instance	100	io1 (2000)	us-east-1a	
PHP-LB-Standard	vol-53af9110	assign to instance	100	standard	us-east-1a	
+ Unregistered Volumes						

2. On the volume's details page, select the appropriate instance, specify the volume's name and mount point, and click **Save** to attach the volume to the instance.

## Volume PHP-LB-PIOPS

Name	PHP-LB-PIOPS
EC2 Volume ID	vol-0faf914c
Mount point	/vol/mountpoint
Availability Zone	us-east-1a
Instance	<input style="width: 150px; height: 20px; border: none; background-color: #f0f0f0; border-bottom: 2px solid #0072bc; color: #0072bc; font-weight: bold; font-size: 10px; padding: 2px 5px; margin-bottom: 5px;" type="button" value="PHP App Server"/>
Status	<input style="width: 150px; height: 20px; border: none; background-color: #f0f0f0; border-bottom: 2px solid #0072bc; color: #0072bc; font-weight: bold; font-size: 10px; padding: 2px 5px; margin-bottom: 5px;" type="button" value="php-app1"/>
Size	100 GiB
Device	-
Volume Type	io1
IOPS	2000
Snapshot ID	-
OpsWorks ID	a402f9f9-6814-403d-8b2d-dfee98950e9c

[Cancel](#) [Save](#)

### Important

If you have assigned an external in-use volume to your instance, you must use the Amazon EC2 console, API, or CLI to unassign it from the original instance or the start process will fail.

You can also use the details page to move an assigned Amazon EBS volume to another instance in the stack.

### To move an Amazon EBS volume to another instance

1. Ensure that both instances are in the offline state.
2. On the **Resources** page, click **Volumes** and then click **edit** in the volume's **Actions** column.
3. Do one of the following:
  - To move the volume to another instance in the stack, select the appropriate instance from the **Instance** list and click **Save**.
  - To move the volume to an instance in another stack, [deregister the volume \(p. 228\)](#), [register the volume \(p. 219\)](#) with the new stack, and [attach it \(p. 222\)](#) to the news instance.

## Associating Elastic IP Addresses with an Instance

You can associate a registered Elastic IP address with an instance and move it from one instance to another, including instances in other stacks. The instances can be either online or offline.

### To associate an Elastic IP address with an instance

1. On the **Resources** page, click **associate with instance** in the appropriate address's **Instance** column.

## Resources

[Show Unregistered Elastic IPs](#)

Address	Name	Instance	Public DNS	Actions
23.21.119.187	-	associate with instance	-	<a href="#">edit</a>
<a href="#">+ Unregistered Elastic IPs</a>				

2. On the address's details page, select the appropriate instance, specify the address's name, and click **Save** to associate the address with the instance.

### Elastic IP 23.21.119.187

IP	23.21.119.187
Name	<input type="text" value="PHP-EIP"/>
Region	us-east-1
Domain	standard
Stack	MyStack <a href="#">change..</a>
Instance	<input style="width: 150px; height: 25px; border: 1px solid #ccc;" type="button" value="Select the instance the Elastic IP should be associated with."/> <div style="border: 1px solid #ccc; padding: 5px; width: 150px; margin-top: 5px;"> <div style="font-size: small; margin-bottom: 5px;">PHP App Server</div> <div style="font-size: small; background-color: #f0f0f0; padding: 2px;">php-app1</div> <div style="font-size: small; background-color: #f0f0f0; padding: 2px;">php-app2</div> <div style="font-size: small; background-color: #f0f0f0; padding: 2px;">php-app3</div> <div style="font-size: small; margin-top: 5px;">Not associated</div> </div>
<a href="#">Cancel</a> <span style="background-color: #0070C0; color: white; padding: 2px 10px; border-radius: 5px; border: none; font-weight: bold;">Save</span>	

#### Note

If the Elastic IP address is currently associated with another online instance, AWS OpsWorks automatically reassigns the address to the new instance.

You can also use the details page to move an associated Elastic IP address to another instance.

#### To move an Elastic IP address to another instance

1. On the **Resources** page, click **Elastic IPs** and click **edit** in the address's **Actions** column.
2. Do one of the following:
  - To move the address to another instance in the stack, select the appropriate instance from the **Instance** list and click **Save**.
  - To move the address to an instance in another stack, click **change** in the **Stack** settings to see a list of the available stacks. Select a stack from the **Stack** list and an instance from the **Instance** list. Then click **Save**.

### Elastic IP PHP-EIP1

IP	54.221.232.99
Name	<input type="text" value="PHP-EIP1"/>
Region	us-east-1
Domain	standard
Stack	MyStack <a href="#">change..</a>
Instance	<input style="width: 150px; height: 25px; border: 1px solid #ccc;" type="button" value="php-app1 [current]"/>

## Attaching Amazon RDS Instances to an App

You can attach an Amazon RDS instance to one or more apps.

### To attach an Amazon RDS instance to an app

1. On the **Resources** page, click **Add app** in the appropriate instance's **Apps** column.

Resources							<a href="#">Show Unregistered RDS DB instances</a>
Volumes	Elastic IPs	RDS					Search
Instance Identifier	Engine	Apps	Type	Multi-AZ	AZ	Actions	
opsinstance1	mysql	<a href="#">Add app</a>	t1.micro	No	us-east-1d	 <a href="#">edit</a>	
<a href="#">+ Unregistered RDS DB instances</a>							

2. Use the **Add App** page to attach the Amazon RDS instance. For more information, see [Adding Apps \(p. 125\)](#).

Because an Amazon RDS can be attached to multiple apps, there is no special procedure for moving the instance from one app to another. Just edit the first app to remove the RDS instance or edit the second app to add the RDS instance. For more information, see [Editing Apps \(p. 134\)](#).

## Detaching Resources

When you no longer need an attached resource, you can detach it. This resource remains registered with the stack and can be attached elsewhere.

### Topics

- [Unassigning Amazon EBS Volumes \(p. 226\)](#)
- [Disassociating Elastic IP Addresses \(p. 227\)](#)
- [Detaching Amazon RDS Instances \(p. 227\)](#)

## Unassigning Amazon EBS Volumes

Use the following procedure to unassign an Amazon EBS volume from its instance.

### To unassign an Amazon EBS volume

1. Ensure that the instance is in the offline state.
2. On the **Resources** page, click **Volumes** and click volume name.
3. On the volume's details page, click **Unassign**.

## Volume PHP-LB-PIOPS

[Edit](#) [Unassign](#)

Volumes are the block level storage associated with your instance. [Learn more.](#)

### Settings

Name	PHP-LB-PIOPS
EC2 Volume ID	vol-0faf914c
Mount point	/vol/mountpoint
Availability Zone	us-east-1a
Instance	<a href="#">php-app1</a> ●
Status	<span>available</span>
Size	100 GiB
Device	/dev/sdi
Volume Type	io1
IOPS	2000
Snapshot ID	—
OpsWorks ID	a402f9f9-6814-403d-8b2d-dfee98950e9c

## Disassociating Elastic IP Addresses

Use the following procedure to disassociate an Elastic IP address from its instance.

### To disassociate an Elastic IP address

1. On the **Resources** page, click **Elastic IPs** and click **edit** in the address's **Actions** column.
2. On the address's details page, click **Disassociate**.

## Elastic IP PHP-Vol2

[Edit](#) [Disassociate](#)

Elastic IPs are static IP addresses for your instance. [Learn more.](#)

### Settings

IP	23.21.119.187
Name	PHP-Vol2
Region	us-east-1
Domain	standard
Instance	<a href="#">php-app1</a> ●

## Detaching Amazon RDS Instances

Use the following procedure to detach an Amazon RDS from an app.

### To detach an Amazon RDS instance

1. On the **Resources** page, click **RDS** and click the appropriate app in the **Apps** column.
2. Click **Edit** and edit the app configuration to detach the instance. For more information, see [Editing Apps \(p. 134\)](#).

**Note**

This procedure detaches an Amazon RDS from a single app. If the instance is attached to multiple apps, you must repeat this procedure for each app.

## Deregistering Resources

If you no longer need to have a resource registered with a stack, you can deregister it. Deregistration does not delete the resource from your account; it remains there and can be registered with another stack or used outside AWS OpsWorks. If you want to delete the resource entirely, you have two options:

- If an Elastic IP or Amazon EBS resource is attached to an instance, you can delete the resource when you delete the instance.

Go to the **Instances** page, click **delete** in the instance's **Actions** column, and then select **Delete instance's EBS volumes** or **Delete the instance's Elastic IP**.

- Deregister the resource and then use the Amazon EC2 or Amazon RDS console, API, or CLI to delete it.

**Topics**

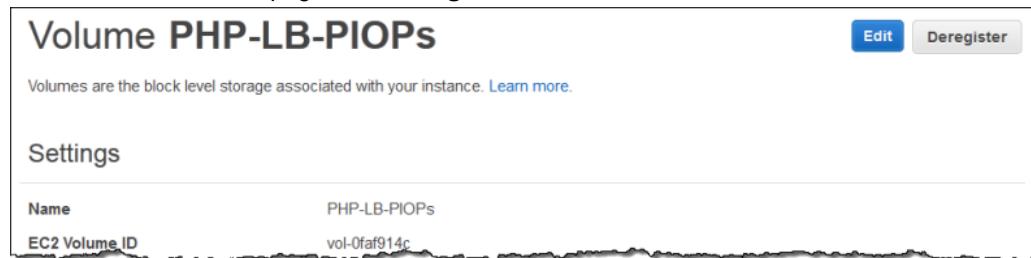
- [Deregistering Amazon EBS Volumes \(p. 228\)](#)
- [Deregistering Elastic IP Addresses \(p. 228\)](#)
- [Deregistering Amazon RDS Instances \(p. 229\)](#)

## Deregistering Amazon EBS Volumes

Use the following procedure to deregister an Amazon EBS volume.

**To deregister an Amazon EBS volume**

1. If the volume is attached to an instance, unassign it, as described in [Unassigning Amazon EBS Volumes \(p. 226\)](#).
2. On the **Resources** page, click the volume name in the **Name** column.
3. On the volume's details page, click **Deregister**.



## Deregistering Elastic IP Addresses

Use the following procedure to deregister an Elastic IP address.

**To deregister an Elastic IP address**

1. If the address is associated with an instance, disassociate it, as described in [Disassociating Elastic IP Addresses \(p. 227\)](#).

2. On the **Resources** page, click **Elastic IPs** and then click the IP address in the **Address** column.
3. On the address's details page, click **Deregister**.

## Elastic IP PHP-Vol2

[Edit](#) [Deregister](#)

Elastic IPs are static IP addresses for your instance. [Learn more](#).

### Settings

IP	23.21.119.187
Name	PHP-Vol2
Region	us-east-1
Domain	standard
Instance	<i>associate with instance</i>

### Tip

If you simply want to register an Elastic IP address with a different stack, you must deregister it from its current stack and then register it with the new stack. However, you can move an attached Elastic IP address to an instance in another stack directly. For more information, see [Attaching and Moving Resources \(p. 222\)](#).

## Deregistering Amazon RDS Instances

Use the following procedure to deregister an Amazon RDS instance.

### To deregister an Amazon RDS instance

1. If the instance is associated with an app, detach it, as described in [Detaching Resources \(p. 226\)](#).
2. On the **Resources** page, click **RDS** and then instance's name.
3. On the instance's details page, click **Deregister**.

**RDS opsinstance1**

[Edit](#) [Deregister](#)

Amazon Relational Database Service (RDS) makes it easy to set up, operate, and scale a relational database in the cloud. [Learn more](#).

Settings	Configuration details
Instance Identifier <a href="#">opsinstance1</a>	DB Engine mysql (5.6.13)

# Customizing AWS OpsWorks

---

AWS OpsWorks built-in layers provide standard functionality that is sufficient for many purposes. However, for many projects, you might encounter one of the following:

- A built-in layer's standard configuration is adequate but not ideal; you would like to optimize it for your particular requirements.

For example, you might want to tune a Static Web Server layer's Nginx server configuration by specifying your own values for settings such as the maximum number of worker processes or the `keepalive-timeout` value.

- A built-in layer's functionality is fine, but you want to extend it by installing additional packages or running some custom installation scripts.

For example, you might want to extend a PHP App Server layer by also installing a Redis server.

- You have requirements that aren't handled by any of the built-in layers.

For example, AWS OpsWorks does not include built-in layers for some popular database servers. You can create a custom layer that installs those servers on the layer's instances.

AWS OpsWorks provides a variety of ways to customize layers to meet your specific requirements. The following examples are listed in order of increasing complexity and power:

- Use custom JSON to override default AWS OpsWorks settings.
- Implement a custom Chef cookbook with an attributes file that overrides the default AWS OpsWorks settings.
- Implement a custom Chef cookbook with a template that overrides or extends a default AWS OpsWorks template.
- Implement a custom Chef cookbook with a simple recipe that runs a shell script.
- Implement a custom Chef cookbook with recipes that perform tasks such as creating and configuring directories, installing packages, creating configuration files, deploying apps, and so on.

## Important

You cannot override a built-in recipe by implementing a custom recipe with the same cookbook and recipe name. For each lifecycle event, AWS OpsWorks always runs the built-in recipes first, followed by any custom recipes. Because Chef does not run a recipe with the same cookbook and recipe name twice, the built-in recipe takes precedence and the custom recipe is not executed.

The following sections describe how to customize an AWS OpsWorks layer:

- AWS OpsWorks stores configuration data and a variety of other information as attributes in a JSON structure on each instance. [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#) describes how the structure's attribute values are determined, and how you can override them.
- The remaining sections describe various ways to customize a layer.

**Note**

Because many of the techniques use custom cookbooks, you should first read [Cookbooks and Recipes \(p. 153\)](#) if you are not already familiar with cookbook implementation.

**Topics**

- [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#)
- [Extending AWS OpsWorks Configuration Files Using Custom Templates \(p. 236\)](#)
- [Extending a Built-in Layer \(p. 237\)](#)
- [Creating a Custom Tomcat Server Layer \(p. 240\)](#)
- [Stack Configuration and Deployment JSON \(p. 262\)](#)

## Customizing AWS OpsWorks Configuration by Overriding Attributes

Recipes and templates depend on a variety of Chef attributes for instance or stack-specific information such as layer configurations or application server settings. These attributes have several sources:

- **Custom JSON**—You can optionally specify custom JSON attributes when you create, update, or clone a stack, or when you deploy an app.
- **Stack configuration JSON**—AWS OpsWorks creates this structure to hold stack configuration information that is determined by the service, including the information that you specify through the console settings.
- **Deployment JSON**—AWS OpsWorks incorporates deployment-related attributes into the stack configuration JSON for Deploy events.
- **Cookbook attributes**—Built-in and custom cookbooks usually include one or more [attributes files \(p. 156\)](#), which contain attributes that represent cookbook-specific values such as application server configuration settings.
- **Chef**—Chef's Ohai tool defines attributes that represent a wide variety of system configuration settings, such as CPU type and installed memory.

For a complete list of stack configuration and deployment JSON, and built-in cookbook attributes, see [Appendix C: AWS OpsWorks Attribute Reference \(p. 378\)](#). For more information about Ohai attributes, see [Ohai](#).

When a [lifecycle event \(p. 176\)](#) such as Deploy or Configure occurs, or you run a [stack command \(p. 52\)](#) such as `execute_recipes` or `update_packages`, AWS OpsWorks does the following:

- Sends a corresponding command to the agent on each affected instance.  
The agent runs the appropriate recipes. For example, for a Deploy event, the agent runs the built-in Deploy recipes, followed by any custom Deploy recipes.
- Merges any custom JSON and deployment JSON with the stack configuration JSON and downloads it to the instances.

The attributes from custom JSON, stack configuration and deployment JSON, cookbook attributes, and Ohai attributes are merged into a *node object*, which supplies attribute values to recipes. An instance is essentially stateless as far as stack configuration attributes are concerned, including any custom JSON. When you run a deployment or stack command, the associated recipes use the stack configuration attributes that were downloaded with the command.

## Attribute Precedence

If all attributes are unique, Chef simply incorporates them into the node object. However, any attribute source can define any attribute, so it is possible for the same attribute to have multiple definitions with different values. For example, the built-in apache2 cookbook defines `node[:apache][:keepalive]`, but you could also define that attribute in custom JSON or in a custom cookbook. The node object can have only one instance of each attribute. If an attribute has multiple definitions, they are evaluated in an order that is described later, and the node object receives the definition with the highest precedence.

An attribute is defined as follows:

```
node.type[:attribute][:sub_attribute][:...]=value
```

If an attribute has multiple definitions, the type determines which definition has precedence, and that definition is incorporated into the node object. AWS OpsWorks uses the following attribute types:

- **default**—This is the most common type, and it essentially means "use this value if the attribute hasn't already been defined." If all definitions of an attribute are `default` type, the first definition in the evaluation order has precedence and subsequent values are ignored. Note that AWS OpsWorks sets all stack configuration and deployment JSON attribute definitions to `default` type.
- **normal**—Attributes with this type override any `default` or `normal` attributes that were defined earlier in the evaluation order. For example, if the first attribute is from a built-in cookbook and has a `default` type and the second is a user-defined attribute with a `normal` type, the second definition has precedence.
- **set**—This is a deprecated type that you might see in older cookbooks. It has been superseded by `normal`, which has the same precedence.

### Note

Chef supports several additional attribute types, including an `automatic` type that takes precedence over all other attribute definitions. The attribute definitions generated by Chef's Ohai tool are all `automatic` types, so they are effectively read-only. This isn't usually an issue, because there is no reason to override them and they are distinct from AWS OpsWorks' attributes. However, you should be careful to name your custom cookbook attributes so they are distinct from the Ohai attributes. For more information, see [About Attribute Files](#).

Here are the steps that incorporate the various attribute definitions into the node object:

1. Merge any custom stack configuration JSON attributes into the stack configuration and deployment JSON.

Custom JSON attributes can be set for the stack, or for a particular deployment. They are first in the evaluation order and are effectively `normal` types. If one or more stack configuration JSON attributes are also defined in custom JSON, the custom JSON values take precedence. Otherwise AWS OpsWorks simply incorporates the custom JSON attributes into the stack configuration JSON.

2. Merge any custom deployment JSON into the stack configuration and deployment JSON.

Custom deployment JSON is also effectively a `normal` type, so it takes precedence over built-in and custom stack configuration JSON and built-in deployment JSON.

3. Pass the stack configuration and deployment JSON to the instance and incorporate its attributes into the instance's node object.

4. Merge the instance's built-in cookbook attributes into the node object.

The built-in cookbook attributes are all `default` types. If the one or more built-in cookbook attributes are also defined in the stack configuration and deployment JSON—typically because you defined them with custom JSON—the stack configuration definitions take precedence over the built-in cookbook definitions. All other built-in cookbook attributes are simply incorporated into the node object.

5. Merge the instance's custom cookbook attributes into the node object.

Custom cookbook attributes are usually either `normal` or `default` types. Unique attributes are incorporated into the node object. If any custom cookbook attributes are also defined in Steps 1–3 (typically because you defined them with custom JSON), precedence depends on the custom cookbook attribute's type:

- Attributes defined in Steps 1–3 take precedence over custom cookbook `default` attributes.
- Custom cookbook `normal` attributes take precedence over definitions from Steps 1–3.

**Important**

Do not use custom cookbook `default` attributes to override stack configuration or built-in cookbook attributes. Because custom cookbook attributes are evaluated last, the `default` attributes have the lowest precedence, and cannot override anything.

## Overriding Attributes Using Custom JSON

The simplest way to override an AWS OpsWorks attribute is to define it in custom JSON, which takes precedence over stack configuration and deployment JSON attributes as well as built-in and custom cookbook `default` attributes. For more information, see [Attribute Precedence \(p. 232\)](#).

**Important**

You should override stack configuration and deployment JSON attributes with care. For example overriding attributes in the `opsworks` namespace can interfere with the built-in recipes. For more information, see [Stack Configuration and Deployment JSON \(p. 262\)](#).

You can also use custom JSON to define unique attributes, typically to pass data to your custom recipes. The attributes are simply incorporated into the node object, and recipes can reference them by using the standard Chef node syntax.

## How to Specify Custom JSON

To use custom JSON to override an attribute value, you must first determine the attribute's fully qualified attribute name. You then create a JSON object that contains the attributes you want to override, set to your preferred values. For convenience, [Appendix C: AWS OpsWorks Attribute Reference \(p. 378\)](#) documents commonly used stack configuration, deployment, and built-in cookbook attributes, including their fully qualified names.

The object's parent-child relationships must correspond to the appropriate fully qualified Chef nodes. For example, suppose you want to change the following Apache attributes:

- The attribute, whose node is `node[ :apache ][ :keepalivetimeout ]` (p. 402) and has a default value of 3.
- The logrotate `schedule` (p. 403) attribute, whose node is `node[ :apache ][ :logrotate ][ :schedule ]`, and has a default value of "daily".

To override the attributes and set the values to 5 and "weekly", respectively, you would use the following custom JSON:

```
{  
    "apache" : {  
        "keepalivetimeout" : 5,  
        "logrotate" : {  
            "schedule" : "weekly"  
        }  
    }  
}
```

## When to Specify Custom JSON

You can specify a custom JSON structure for the following tasks:

- [Create a new stack \(p. 41\)](#)
- [Update a stack \(p. 50\)](#)
- [Run a stack command \(p. 50\)](#)
- [Clone a stack \(p. 51\)](#)
- [Deploy an app \(p. 132\)](#)

For each task, AWS OpsWorks merges the custom JSON with the stack configuration and deployment JSON and sends it to the instances, to be merged into the node object. However, note the following:

- If you specify custom JSON when you create, clone, or update a stack, it is merged into the stack configuration and deployment JSON and is sent to instances for all subsequent lifecycle events and stack commands.
- If you specify custom JSON for a deployment, it is merged into the stack configuration and deployment JSON only for the corresponding event.

If you want to use that JSON for subsequent deployments, you must explicitly specify it again.

It is important to remember that attributes only affect the instance when they are used by recipes. If you override an attribute value but no subsequent recipes reference the attribute, the change has no effect. You must either ensure that the custom JSON is sent before the associated recipes run, or ensure that the appropriate recipes are re-run.

## Custom JSON Best Practices

You can use custom JSON to override any AWS OpsWorks attribute, but manually entering the information is somewhat cumbersome, and it is not under any sort of source control. Custom JSON is best used for the following purposes:

- When you want to override only a small number of attributes, and you do not otherwise need to use custom cookbooks.

With Custom JSON, you can avoid the overhead of setting up and maintaining a cookbook repository just to override a couple of attributes.

- Sensitive values, such as passwords or authentication keys.

Cookbook attributes are stored in a repository, so any sensitive information is at some risk of being compromised. Instead, define attributes with dummy values and use custom JSON to set the real values.

- Values that are expected to vary.

For example, a recommended practice is to have your production stack supported by separate development and staging stacks. Suppose that these stacks support an application that accepts payments. If you use custom JSON to specify the payment endpoint, you can specify a test URL for your staging stack. When you are ready to migrate an updated stack to your production stack, you can use the same cookbooks and use custom JSON to set the payment endpoint to the production URL.

- Values that are specific to a particular stack or deployment command.

## Overriding AWS OpsWorks Attributes Using Custom Cookbook Attributes

Custom JSON is a convenient way to override AWS OpsWorks stack configuration and built-in cookbook attributes, but it has some limitations. In particular, you must enter custom JSON manually for each use, so you have no robust way to manage the definitions. A better approach is often to use custom cookbook attribute files to override built-in attributes. Doing so allows you to place the definitions under source control.

The procedure for using custom attribute files to override AWS OpsWorks definitions is straightforward.

### To override AWS OpsWorks attribute definitions

1. Set up a cookbook repository, as described in [Cookbooks and Recipes \(p. 153\)](#).
2. Create a cookbook with the same name as the built-in cookbook that contains the attributes that you want to override. For example, to override the Apache attributes, the cookbook should be named apache2.
3. Add an `attributes` folder to the cookbook and add a file to that folder named `customize.rb`.
4. Add an attribute definition to the file for each of the built-in cookbook's attributes that you want to override, set to your preferred value. The attribute must be a `normal` type or higher and have exactly the same node name as the corresponding AWS OpsWorks attribute. For a detailed list of AWS OpsWorks attributes, including node names, see [Appendix C: AWS OpsWorks Attribute Reference \(p. 378\)](#). For more information on attributes and attributes files, see [About Attribute Files](#).

#### Important

Your attributes must be `normal` type to override AWS OpsWorks attributes; `default` types do not have precedence. For example, if your `customize.rb` file contains a `default[:apache][:keepalivetimeout] = 5` attribute definition, the corresponding attribute in the built-in `apache.rb` attributes file is evaluated first, and takes precedence. For more information, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

5. Repeat Steps 2 – 4 for each built-in cookbook with attributes that you want to override.
6. Enable custom cookbooks for your stack and provide the information required for AWS OpsWorks to download your cookbooks to the stack's instances. For more information, see [Installing Custom Cookbooks \(p. 171\)](#).

The node object used by subsequent lifecycle events, deploy commands, and stack commands will now contain your attribute definitions instead of the AWS OpsWorks values.

For example, to override the built-in Apache `keepalivetimeout` and `logrotate schedule` settings discussed in [How to Specify Custom JSON \(p. 233\)](#), add an `apache2` cookbook to your repository and add a `customize.rb` file to the cookbook's `attributes` folder with the following contents.

```
normal[:apache][:keepalivetimeout] = 5
normal[:apache][:logrotate][:schedule] = 'weekly'
```

**Important**

You should not override AWS OpsWorks attributes by modifying a copy of the associated built-in attributes file. If, for example, you copy `apache.rb` to your `apache2/attributes` folder and modify some of its settings, you essentially override every attribute in the built-in file. Recipes will use the attribute definitions from your copy and ignore the built-in file. If AWS OpsWorks later modifies the built-in attributes file, recipes will not have access to the changes unless you manually update your copy.

To avoid this situation, all built-in cookbooks contain an empty `customize.rb` attributes file, which is required in all modules through an `include_attribute` directive. By overriding attributes in your copy of `customize.rb`, you affect only those specific attributes. Recipes will obtain any other attribute values from the built-in attributes files, and automatically get the current values of any attributes that you have not overridden.

This approach helps you to keep the number of attributes in your cookbook repository small, which reduces your maintenance overhead and makes future upgrades easier to manage.

## Extending AWS OpsWorks Configuration Files Using Custom Templates

AWS OpsWorks uses templates to create files such as configuration files, which typically depend on attributes for many of the settings. If you use custom JSON or custom cookbook attributes to override the AWS OpsWorks definitions, your preferred settings are incorporated into the configuration files in place of the AWS OpsWorks settings. However, AWS OpsWorks does not necessarily specify an attribute for every possible configuration setting; it accepts the defaults for some settings and hardcodes others directly in the template. You can't use custom JSON or custom cookbook attributes to specify preferred settings if there is no corresponding AWS OpsWorks attribute.

You can extend the configuration file to include additional configuration settings by creating a custom template. You can then add whatever configuration settings or other content you need to the file, and override any hardcoded settings. For more information on templates, see [Templates \(p. 158\)](#).

**Note**

You can override any built-in template *except* `opsworks-agent.monitrc.erb`.

### To create a custom template

1. Create a cookbook with the same structure and directory names as the built-in cookbook. Then, create a template file in the appropriate directory with the same name as the built-in template that you want to customize. For example, to use a custom template to extend the Apache `httpd.conf` configuration file, you must implement an `apache2` cookbook in your repository and your template file must be `apache2/templates/default/apache.conf.erb`. Using exactly the same names allows AWS OpsWorks to recognize the custom template and use it instead of the built-in template.

The simplest approach is to just copy the built-in template file from the [built-in cookbook's GitHub repository](#) to your cookbook and modify it as needed.

**Important**

Do not copy any files from the built-in cookbook except for the template files that you want to customize. Copies of other types of cookbook file, such as recipes, create duplicate Chef resources and can cause errors.

The cookbook can also include custom attributes, recipes, and related files, but their file names should not duplicate built-in file names.

2. Customize the template file to produce a configuration file that meets your requirements. You can add more settings, delete existing settings, replace hardcoded attributes, and so on.

3. If you haven't done so already, edit the stack settings to enable custom cookbooks and specify your cookbook repository. For more information, see [Installing Custom Cookbooks \(p. 171\)](#).

You don't have to implement any recipes or [add recipes to the layer configuration \(p. 177\)](#) to override a template. AWS OpsWorks always runs the built-in recipes. When it runs the recipe that creates the configuration file, it will automatically use your custom template instead of the built-in template.

**Note**

If AWS OpsWorks makes any changes to the built-in template, your custom template might become out of sync and no longer work correctly. For example, suppose your template refers to a dependent file, and the file name changes. AWS OpsWorks doesn't make such changes often, and when a template does change, it lists the changes and gives you the option of upgrading to a new version. You should monitor the AWS OpsWorks repository for changes, and manually update your template as needed.

## Extending a Built-in Layer

Sometimes, you need to customize a built-in layer beyond what can be handled by modifying AWS OpsWorks attributes or customizing templates. For example, suppose you need to create symlinks, set file or folder modes, install additional packages, and so on. In that case, you will need to implement one or more recipes to handle the customization tasks.

**Topics**

- [Using Recipes to Run Scripts \(p. 237\)](#)
- [Using Chef Deployment Hooks \(p. 238\)](#)
- [Running Cron Jobs \(p. 239\)](#)
- [Installing and Configuring Packages \(p. 240\)](#)

## Using Recipes to Run Scripts

If you already have a script that performs the required customization tasks, the simplest approach to extending a layer is often to implement a simple recipe to run the script. You can then assign the recipe to the appropriate lifecycle events, typically Setup or Deploy, or use the `execute_recipes` stack command to run the recipe manually.

The following example runs a shell script but you can use the same approach for other types of script.

```
cookbook_file "/tmp/lib-installer.sh" do
  source "lib-installer.sh"
  mode 0755
end

execute "install my lib" do
  command "sh /tmp/lib-installer.sh"
end
```

The `cookbook_file` resource represents a file that is stored in a subdirectory of a cookbook's `files` directory, and transfers the file to a specified location on the instance. This example transfers a shell script, `lib-installer.sh`, to the instance's `/tmp` directory and sets the file's mode to 0755. For more information, see [cookbook\\_file](#).

The `execute` resource represents a command, such as a shell command. This example runs `lib-installer.sh`. For more information, see [execute](#).

You can also run a script by incorporating it into a recipe. The following example runs a bash script, but Chef also supports Csh, Perl, Python, and Ruby.

```
script "install_something" do
  interpreter "bash"
  user "root"
  cwd "/tmp"
  code <<-EOH
    #insert bash script
  EOH
end
```

The `script` resource represents a script. The example specifies a bash interpreter, sets user to `"root"`, and sets the working directory to `/tmp`. It then runs the bash script in the `code` block, which can include as many lines as required. For more information, see [script](#).

## Using Chef Deployment Hooks

You can customize deployment by implementing a recipe to perform the required customization and assign it to the appropriate layer's Deploy event. An alternative and sometimes simpler approach—especially if you don't need to implement a cookbook for other purposes—is to use Chef deployment hooks to run your customization code. In addition, custom Deploy recipes run after the deployment has already been performed by the built-in recipes. Deployment hooks allow you to interact during a deployment, for example, after the app's code is checked out of the repository but before Apache is restarted.

Chef deploys apps in four stages:

- **Checkout**—Downloads the files from the repository
- **Migrate**—Runs a migration, as required
- **Symlink**—Creates symlinks
- **Restart**—Restarts the application

Chef deployment hooks provide a simple way to customize a deployment by optionally running a user-supplied Ruby application after each stage completes. To use deployment hooks, implement one or more Ruby applications and place them in your app's `/deploy` directory. The application must have one of the following names, which determines when it runs.

- `before_migrate.rb` runs after the Checkout stage is complete but before Migrate.
- `before_symlink.rb` runs after the Migrate stage is complete but before Symlink.
- `before_restart.rb` runs after the Symlink stage is complete but before Restart.
- `after_restart.rb` runs after the Restart stage is complete.

Chef deployment hooks can access the node object just like recipes. For more information on how to use deployment hooks and what information a hook can access, see [Deploy Phases](#).

## Running Cron Jobs

A cron job directs the cron daemon to run one or more commands on a specified schedule. For example, suppose your stack supports a PHP e-commerce application. You can set up a cron job to have the server send you a sales report at a specified time every week. For more information, see [cron](#).

Instead of manually setting up cron jobs on your instances, you can have AWS OpsWorks handle the task automatically. The following procedure describes how to set up a cron job on a PHP App Server layer's instances, but you can use the same approach with any layer.

### To set up a cron job on a layer's instances

1. Implement a cookbook with a recipe with a cron resource that sets up the job. The example assumes that the recipe is named `cronjob.rb`; the implementation details are described later. For more information on cookbooks and recipes, see [Cookbooks and Recipes \(p. 153\)](#).
2. Install the cookbook on your stack. For more information, see [Installing Custom Cookbooks \(p. 171\)](#).
3. Have AWS OpsWorks run the recipe automatically on the layer's instances by assigning it to the following lifecycle events. For more information, see [Automatically Running Recipes \(p. 177\)](#).
  - **Setup** – Assigning `cronjob.rb` to this event directs AWS OpsWorks to run the recipe on all new instances.
  - **Deploy** – Assigning `cronjob.rb` to this event directs AWS OpsWorks to run the recipe on all online instances when you deploy or redeploy an app to the layer.

You can also manually run the recipe on online instances by using the `Execute Recipes` stack command. For more information, see [Run Stack Commands \(p. 52\)](#).

The following is the `cronjob.rb` example, which sets up a cron job to run a user-implemented PHP application once a week that collects the sales data from the server and mails a report. For more examples of how to use a cron resource, see [cron](#).

```
crond "job_name" do
  hour "1"
  minute "10"
  weekday "6"
  command "cd /srv/www/myapp/current && php .lib/mailing.php"
end
```

`crond` is a Chef resource that represents a cron job. When AWS OpsWorks runs the recipe on an instance, the associated provider handles the details of setting up the job.

- `job_name` is a user-defined name for the cron job, such as `weekly_report`.
- `hour/minute/weekday` specify when the commands should run. This example runs the commands every Saturday at 1:10 AM.
- `command` specifies the commands to be run.

This example runs two commands. The first navigates to the `/srv/www/myapp/current` directory. The second runs the user-implemented `mailing.php` application, which collects the sales data and sends the report.

If your stack has multiple application servers, assigning `cronjob.rb` to the PHP App Server layer's lifecycle events might not be an ideal approach. For example, the recipe runs on all of the layer's instances, so

you will receive multiple reports. A better approach is to use a custom layer to ensure that only one server sends a report.

#### To run a recipe on just one of a layer's instances

1. Create a custom layer called, for example, PHPAdmin and assign `cronjob.rb` to its Setup and Deploy events. Custom layers don't necessarily have to do very much. In this case, PHPAdmin just runs one custom recipe on its instances.
2. Assign one of the PHP App Server instances to AdminLayer. If an instance belongs to more than one layer, AWS OpsWorks runs each layer's built-in and custom recipes.

Because only one instance belongs to the PHP App Server and PHPAdmin layers, `cronjob.rb` runs only on that instance and you receive just one report.

## Installing and Configuring Packages

The built-in layers support only certain packages. For more information, see [Layers \(p. 57\)](#). You can install other packages, such as a Redis server, by implementing custom recipes to handle the associated setup, configuration, and deployment tasks. In some cases, the best approach is to extend a built-in layer to have it install the package on its instances alongside the layer's standard packages. For example, if you have a stack that supports a PHP application, and you would like to include a Redis server, you could extend the PHP App Server layer to install and configure a Redis server on the layer's instances in addition to a PHP application server.

A package installation recipe typically needs to perform tasks like these:

- Create one or more directories and set their modes.
- Create a configuration file from a template.
- Run the installer to install the package on the instance.
- Start one or more services.

For an example of how to install a Tomcat server, see [Creating a Custom Tomcat Server Layer \(p. 240\)](#). The topic describes how to set up a custom Redis layer, but you could use much the same code to install and configure Redis on a built-in layer. For examples of how to install other packages, see the built-in cookbooks, at <https://github.com/aws/opsworks-cookbooks>.

## Creating a Custom Tomcat Server Layer

The simplest way to use nonstandard packages on AWS OpsWorks instances is to [extend an existing layer \(p. 240\)](#). However, this approach installs and runs both the standard and nonstandard packages on the layer's instances, which is not always desirable. A somewhat more demanding but more powerful approach is to implement a custom layer, which gives you almost complete control over the layer's instances, including the following:

- Which packages are installed
- How each package is configured
- How to deploy apps from a repository to the instance

Whether using the console or API, you create and manage a custom layer much like any other layer, as described in [Custom AWS OpsWorks Layers \(p. 96\)](#). However, a custom layer's built-in recipes perform only some very basic tasks, such as installing a Ganglia client to report metrics to a Ganglia master. To make a custom layer's instances more than minimally functional, you must implement one or more custom

cookbooks with Chef recipes and related files to handle the tasks of installing and configuring packages, deploying apps, and so on. You don't necessarily have to implement everything from scratch, though. For example, if you store applications in one of the standard repositories, you can use the built-in deploy recipes to handle much of the work of installing the applications on the layer's instances.

The following walkthrough describes how to implement a custom layer that supports a Tomcat application server. The layer is based on a custom cookbook named Tomcat, which includes recipes to handle package installation, deployment, and so on. The walkthrough includes excerpts from the Tomcat cookbook. You can download the complete cookbook from its [GitHub repository](#). If you are not familiar with [Opscode Chef](#), you should first read [Cookbooks and Recipes \(p. 153\)](#).

**Note**

AWS OpsWorks includes a full-featured [Java App Server layer \(p. 82\)](#) for production use. The purpose of the Tomcat cookbook is to show how to implement custom layers, so it supports only a limited version of Tomcat that does not include features such as SSL. For an example of a full featured implementation, see the built-in cookbook at [opsworks-cookbooks](#).

The Tomcat cookbook supports a custom layer whose instances have the following characteristics:

- They support a Tomcat Java application server with an Apache front end.
- Tomcat is configured to allow applications to use a JDBC `DataSource` object to connect to a separate MySQL instance, which serves as a back end data store.

The cookbook for this project involves several key components:

- [Attributes file \(p. 241\)](#) contains configuration settings that are used by the various recipes.
- [Setup recipes \(p. 242\)](#) are assigned to the layer's Setup lifecycle event. They run after an instance has booted and perform tasks such as installing packages and creating configuration files.
- [Configure recipes \(p. 250\)](#) are assigned to the layer's Configure lifecycle event. They run after any of the stack's instances come online or go offline and handle any required configuration changes.
- [Deploy recipes \(p. 254\)](#) are assigned to the layer's Deploy lifecycle event. They run after the Setup recipes and when you manually deploy an app to install the code and related files on a layer's instances.

The final section, [Create a Stack and Run an Application \(p. 256\)](#), describes how to create a stack that includes a custom layer based on the Tomcat cookbook and how to deploy and run a simple JSP application that displays data from a MySQL database running on an instance that belongs to a separate MySQL layer.

**Note**

The Tomcat cookbook recipes depend on some AWS OpsWorks built-in recipes. To make each recipe's origin clear, this topic identifies recipes using the Chef `cookbookname::recipename` convention.

## Attributes File

Before looking at the recipes, it is useful to first examine the Tomcat cookbook's attributes file, which contains variety of configuration settings that the recipes use. Attributes aren't required; you can simply hardcode these values in your recipes or templates. However, if you define configuration settings using attributes, you can use the AWS OpsWorks console or API to modify the values by using custom JSON, which is simpler and more flexible than rewriting the recipe or template code every time you want to change a setting. This approach allows you, for example, to use the same cookbook for multiple stacks, but configure the Tomcat server differently for each stack. For more information on attributes and how to override them, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

The following example shows the complete attributes file, `default.rb`, which is located in the Tomcat cookbook's `attributes` directory.

```

default['tomcat']['base_version'] = 6
default['tomcat']['port'] = 8080
default['tomcat']['secure_port'] = 8443
default['tomcat']['ajp_port'] = 8009
default['tomcat']['shutdown_port'] = 8005
default['tomcat']['uri_encoding'] = 'UTF-8'
default['tomcat']['unpack_wars'] = true
default['tomcat']['auto_deploy'] = true
case node[:platform]
when 'centos', 'redhat', 'fedora', 'amazon'
  default['tomcat']['java_opts'] = ''
when 'debian', 'ubuntu'
  default['tomcat']['java_opts'] = '-Djava.awt.headless=true -Xmx128m -
XX:+UseConcMarkSweepGC'
end
default['tomcat']['catalina_base_dir'] = "/etc/tomcat#{node['tomcat']['base_version']}"
default['tomcat']['webapps_base_dir'] = "/var/lib/tomcat#{node['tom
cat']['base_version']}/webapps"
default['tomcat']['lib_dir'] = "/usr/share/tomcat#{node['tomcat']['base_ver
sion']}/lib"
default['tomcat']['java_dir'] = '/usr/share/java'
default['tomcat']['mysql_connector_jar'] = 'mysql-connector-java.jar'
default['tomcat']['apache_tomcat_bind_mod'] = 'proxy_http' # or: 'proxy_ajp'
default['tomcat']['apache_tomcat_bind_config'] = 'tomcat_bind.conf'
default['tomcat']['apache_tomcat_bind_path'] = '/tc/'
default['tomcat']['webapps_dir_entries_to_delete'] = %w(config log public tmp)
case node[:platform]
when 'centos', 'redhat', 'fedora', 'amazon'
  default['tomcat']['user'] = 'tomcat'
  default['tomcat']['group'] = 'tomcat'
  default['tomcat']['system_env_dir'] = '/etc/sysconfig'
when 'debian', 'ubuntu'
  default['tomcat']['user'] = "tomcat#{node['tomcat']['base_version']}"
  default['tomcat']['group'] = "tomcat#{node['tomcat']['base_version']}"
  default['tomcat']['system_env_dir'] = '/etc/default'
end

```

The settings themselves are discussed later in the related section. The following notes apply generally:

- All of the node definitions are `default` type, so you can override them with custom JSON.
- The file uses a `case` statement to conditionally set some attribute values based on instance's operating system.

The `platform` node is generated by Chef's Ohai tool and represents the instance's operating system.

## Setup Recipes

Setup recipes are assigned to the layer's Setup [lifecycle](#) (p. 176) event and run after an instance boots. They perform tasks such as installing packages, creating configuration files, and so on. After the Setup recipes finish running, AWS OpsWorks runs the [Deploy recipes](#) (p. 254) to deploy any apps to the new instance.

### Topics

- [tomcat::setup](#) (p. 243)

- [tomcat::install \(p. 244\)](#)
- [tomcat::service \(p. 245\)](#)
- [tomcat::container\\_config \(p. 246\)](#)
- [tomcat::apache\\_tomcat\\_bind \(p. 249\)](#)

## tomcat::setup

The `tomcat::setup` recipe is intended to be assigned to a layer's Setup lifecycle event.

```
include_recipe 'tomcat::install'
include_recipe 'tomcat::service'

service 'tomcat' do
  action :enable
end

# for EBS-backed instances we rely on autofs
bash '(re-)start autofs earlier' do
  user 'root'
  code <<-EOC
    service autofs restart
  EOC
  notifies :restart, resources(:service => 'tomcat')
end

include_recipe 'tomcat::container_config'
include_recipe 'apache2'
include_recipe 'tomcat::apache_tomcat_bind'
```

`tomcat::setup` recipe is largely a metarecipe. It includes a set of dependent recipes that handle most of the details of installing and configuring Tomcat and related packages. The first part of `tomcat::setup` runs the following recipes, which are discussed later:

- The [tomcat::install \(p. 244\)](#) recipe installs the Tomcat server package.
- The [tomcat::service \(p. 245\)](#) recipe sets up the Tomcat service.

The middle part of `tomcat::setup` enables and starts the Tomcat service:

- The Chef `service` resource enables the Tomcat service at boot.
- The Chef `bash` resource runs a Bash script to start the autofs daemon, which is necessary for Amazon EBS-backed instances. The resource then notifies the `service` resource to restart the Tomcat service.

For more information, see: [autofs](#) (for Amazon Linux) or [Autofs](#) (for Ubuntu).

The final part of `tomcat::setup` creates configuration files and installs and configures the front-end Apache server:

- The [tomcat::container\\_config \(p. 246\)](#) recipe creates configuration files.
- The `apache2` recipe (which is shorthand for `apache2::default`) is an AWS OpsWorks built-in recipe that installs and configures an Apache server.
- The [tomcat::apache\\_tomcat\\_bind \(p. 249\)](#) recipe configures the Apache server to function as a front-end for the Tomcat server.

**Tip**

You can often save time and effort by using built-in recipes to perform some of the required tasks. This recipe uses the built in `apache2::default` recipe to install Apache rather than implementing it from scratch. For another example of how to use built-in recipes, see [Deploy Recipes \(p. 254\)](#).

The following sections describe the Tomcat cookbook's Setup recipes in more detail. For more information on the `apache2` recipes, see [opsworks-cookbooks/apache2](#).

## tomcat::install

The `tomcat::install` recipe installs the Tomcat server, the OpenJDK, and a Java connector library that handles the connection to the MySQL server.

```
tomcat_pkgs = value_for_platform(
  ['debian', 'ubuntu'] => {
    'default' => ["tomcat#{node['tomcat']['base_version']}", 'libtcnative-1',
'libmysql-java']
  },
  ['centos', 'redhat', 'fedora', 'amazon'] => {
    'default' => ["tomcat#{node['tomcat']['base_version']}", 'tomcat-native',
'mysql-connector-java']
  },
  'default' => ["tomcat#{node['tomcat']['base_version']}"]
)

tomcat_pkgs.each do |pkg|
  package pkg do
    action :install
  end
end

link ::File.join(node['tomcat']['lib_dir'], node['tomcat']['mysql_connect
or_jar']) do
  to ::File.join(node['tomcat']['java_dir'], node['tomcat']['mysql_connect
or_jar'])
  action :create
end

# remove the ROOT webapp, if it got installed by default
include_recipe 'tomcat::remove_root_webapp'
```

The recipe performs the following tasks:

1. Creates a list of packages to be installed, depending on the instance's operating system.
2. Installs each package in the list.

The Chef `package resource` uses the appropriate provider—`yum` for Amazon Linux and `apt-get` for Ubuntu—to handle the installation. The package providers install OpenJDK as a Tomcat dependency, but the MySQL connector library must be installed explicitly.

3. Uses a Chef `link resource` to create a symlink in the Tomcat server's lib directory to the MySQL connector library in the JDK.

Using the default attribute values, the Tomcat lib directory is `/usr/share/tomcat6/lib` and the MySQL connector library (`mysql-connector-java.jar`) is in `/usr/share/java/`.

The `tomcat::remove_root_webapp` recipe removes the ROOT web application (`/var/lib/tomcat6/webapps/ROOT` by default) to avoid some security issues.

```
ruby_block 'remove the ROOT webapp' do
  block do
    ::FileUtils.rm_rf(::File.join(node['tomcat']['webapps_base_dir'], 'ROOT'),
      :secure => true)
  end
  only_if { ::File.exists?(::File.join(node['tomcat']['webapps_base_dir'],
    'ROOT')) && !::File.symlink?(::File.join(node['tomcat']['webapps_base_dir'],
    'ROOT')) }
end
```

The `only_if` statement ensures that the recipe removes the file only if it exists.

**Tip**

The Tomcat version is specified by the `['tomcat']['base_version']` attribute, which is set to 6 in the attributes file. To install Tomcat 7, you can use custom JSON to override the attribute. Just [edit your stack settings \(p. 50\)](#) and enter the following JSON in the Custom Chef JSON box, or add it to any existing custom JSON:

```
{
  'tomcat' : {
    'base_version' : 7
  }
}
```

The custom JSON attribute overrides the default attribute and sets the Tomcat version to 7. For more information on overriding attributes, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

## tomcat::service

The `tomcat::service` recipe creates the Tomcat service definition.

```
service 'tomcat' do
  service_name "tomcat#{node['tomcat']['base_version']}"

  case node[:platform]
  when 'centos', 'redhat', 'fedora', 'amazon'
    supports :restart => true, :reload => true, :status => true
  when 'debian', 'ubuntu'
    supports :restart => true, :reload => false, :status => true
  end

  action :nothing
end
```

The recipe uses the Chef [service resource](#) to specify the Tomcat service name (`tomcat6`, by default) and sets the `supports` attribute to define how Chef manages the service's restart, reload, and status commands on the different operating systems.

- `true` indicates that Chef can use the init script or other service provider to run the command
- `false` indicates that Chef must attempt to run the command by other means.

Notice that the `action` is set to `:nothing`. For each lifecycle event, AWS OpsWorks initiates a [Chef run](#) to execute the appropriate set of recipes. The Tomcat cookbook follows a common pattern of having a recipe create the service definition, but not restart the service. Other recipes in the Chef run handle the restart, typically by including a `notifies` command in the `template` resources that are used to create configuration files. Notifications are a convenient way to restart a service because they do so only if the configuration has changed. In addition, if a Chef run has multiple restart notifications for a service, Chef restarts the service at most once. This practice avoids problems that can occur when attempting to restart a service that is not fully operational, which is a common source of Tomcat errors.

The Tomcat service must be defined for any Chef run that uses restart notifications. `tomcat::service` is therefore included in several recipes, to ensure that the service is defined for every Chef run. There is no penalty if a Chef run includes multiple instances of `tomcat::service` because Chef ensures that a recipe executes only once per run, regardless of how many times it is included.

## tomcat::container\_config

The `tomcat::container_config` recipe creates configuration files from cookbook template files.

```
include_recipe 'tomcat::service'

template 'tomcat environment configuration' do
  path ::File.join(node['tomcat']['system_env_dir'], "tomcat#{node['tomcat']['base_version']}")
  source 'tomcat_env_config.erb'
  owner 'root'
  group 'root'
  mode 0644
  backup false
  notifies :restart, resources(:service => 'tomcat')
end

template 'tomcat server configuration' do
  path ::File.join(node['tomcat']['catalina_base_dir'], 'server.xml')
  source 'server.xml.erb'
  owner 'root'
  group 'root'
  mode 0644
  backup false
  notifies :restart, resources(:service => 'tomcat')
end
```

The recipe first calls `tomcat::service`, which defines the service if necessary. The bulk of the recipe consists of two [template resources](#), each of which creates a configuration file from one of the cookbook's template files, sets the file properties, and notifies Chef to restart the service.

### Tomcat Environment Configuration File

The first `template` resource uses the `tomcat_env_config.erb` template file to create a Tomcat environment configuration file, which is used to set environment variables such as `JAVA_HOME`. The default file name is the `template` resource's argument. `tomcat::container_config` uses a `path` attribute to override the default value and name the configuration file `/etc/sysconfig/tomcat6` (Amazon Linux) or `/etc/default/tomcat6` (Ubuntu). The `template` resource also specifies the file's owner, group, and mode settings and directs Chef to not create backup files.

If you look at the source code, there are actually three versions of `tomcat_env_config.erb`, each in a different subdirectory of the `templates` directory. The `ubuntu` and `amazon` directories contain the

templates for their respective operating systems. The `default` folder contains a dummy template with a single comment line, which is used only if you attempt to run this recipe on an instance with an unsupported operating system. The `tomcat::container_config` recipe doesn't need to specify which `tomcat_env_config.erb` to use. Chef automatically picks the appropriate directory for the instance's operating system based on rules described in [Location Specificity](#).

The `tomcat_env_config.erb` files for this example consist largely of comments. To set additional environment variables, just uncomment the appropriate lines and provide your preferred values.

**Tip**

Any configuration setting that might change should be defined as an attribute rather than hard-coded in the template. That way, you don't have to rewrite the template to change a setting, you can just override the attribute.

The Amazon Linux template sets only one environment variable, as shown in the following excerpt.

```
...
# Use JAVA_OPTS to set java.library.path for libtcnative.so
#JAVA_OPTS="-Djava.library.path=/usr/lib"

JAVA_OPTS="\${JAVA_OPTS} <%= node['tomcat']['java_opts'] %>

# What user should run tomcat
#TOMCAT_USER="tomcat"
...
```

`JAVA_OPTS` can be used to specify Java options such as the library path. Using the default attribute values, the template sets no Java options for Amazon Linux. You can set your own Java options by overriding the `['tomcat']['java_opts']` attribute, for example, by using custom JSON. For an example, see [Create a Stack \(p. 258\)](#).

The Ubuntu template sets several environment variables, as shown in the following template excerpt.

```
# Run Tomcat as this user ID. Not setting this or leaving it blank will use the
# default of tomcat<%= node['tomcat']['base_version'] %>.
TOMCAT<%= node['tomcat']['base_version'] %>_USER=tomcat<%= node['tom
cat']['base_version'] %>
...
# Run Tomcat as this group ID. Not setting this or leaving it blank will use
# the default of tomcat<%= node['tomcat']['base_version'] %>.
TOMCAT<%= node['tomcat']['base_version'] %>_GROUP=tomcat<%= node['tom
cat']['base_version'] %>
...
JAVA_OPTS=<%= node['tomcat']['java_opts'] %>

<% if node['tomcat']['base_version'].to_i < 7 -%>
# Unset LC_ALL to prevent user environment executing the init script from
# influencing servlet behavior. See Debian bug #645221
unset LC_ALL
<% end -%>
```

Using default attribute values, the template sets the Ubuntu environment variables as follows:

- `TOMCAT6_USER` and `TOMCAT6_GROUP`, which represent the Tomcat user and group, are both set to `tomcat6`.

If you set ['tomcat']['base\_version'] to `tomcat7`, the variable names resolve to `TOMCAT7_USER` and `TOMCAT7_GROUP`, and both are set to `tomcat7`.

- `JAVA_OPTS` is set to `-Djava.awt.headless=true -Xmx128m -XX:+UseConcMarkSweepGC`:
  - Setting `-Djava.awt.headless` to `true` informs the graphics engine that the instance is headless and does not have a console, which addresses faulty behavior of certain graphical applications.
  - `-Xmx128m` ensures that the JVM has adequate memory resources, 128MB for this example.
  - `-XX:+UseConcMarkSweepGC` specifies concurrent mark sweep garbage collection, which helps limit garbage-collection induced pauses.

For more information, see: [Concurrent Mark Sweep Collector Enhancements](#).

- If the Tomcat version is less than 7, the template unsets `LC_ALL`, which addresses a Ubuntu bug.

#### Note

With the default attributes, some of these environment variables are simply set to their default values. However, explicitly setting environment variables to attributes means you can use custom JSON to override the default attributes and provide custom values. For more information on overriding attributes, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

For the complete template files, see the [source code](#).

## Server.xml Configuration File

The second template resource uses `server.xml.erb` to create the [system.xml configuration file](#), which configures the servlet/JSP container. `server.xml.erb` contains no operating system-specific settings, so it is in the template directory's default subdirectory.

The template uses standard settings, but it can create a `system.xml` file for either Tomcat 6 or Tomcat 7. For example, the following code from the template's server section configures the listeners appropriately for the specified version.

```
<% if node['tomcat']['base_version'].to_i > 6 -%>
  <!-- Security listener. Documentation at /docs/config/listeners.html
  <Listener className="org.apache.catalina.security.SecurityListener" />
  -->
<% end -%>
  <!--APR library loader. Documentation at /docs/apr.html -->
  <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEn
gine="on" />
  <!--Initialize Jasper prior to webapps are loaded. Documentation at
  /docs/jasper-howto.html -->
  <Listener className="org.apache.catalina.core.JasperListener" />
  <!-- Prevent memory leaks due to use of particular java/javax APIs-->
  <Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener"
  />
<% if node['tomcat']['base_version'].to_i < 7 -%>
  <!-- JMX Support for the Tomcat server. Documentation at /docs/non-exist
ent.html -->
  <Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" />
<% end -%>
  <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListen
er" />
<% if node['tomcat']['base_version'].to_i > 6 -%>
  <Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListen
```

```
er" />
<% end -%>
```

The template uses attributes in place of hardcoded settings so you can easily change the settings by using custom JSON. For example:

```
<Connector port="<% node['tomcat']['port'] %>" protocol="HTTP/1.1"
           connectionTimeout="20000"
           URIEncoding="<% node['tomcat']['uri_encoding'] %>"
           redirectPort="<% node['tomcat']['secure_port'] %>" />
```

For more information, see the [source code](#).

## tomcat::apache\_tomcat\_bind

The `tomcat::apache_tomcat_bind` recipe enables the Apache server to act as Tomcat's front end, receiving incoming requests and forwarding them to Tomcat and returning the responses to the client. This example uses `mod_proxy` as the Apache proxy/gateway.

```
execute 'enable mod_proxy for apache-tomcat binding' do
  command '/usr/sbin/a2enmod proxy'
  not_if do
    ::File.symlink?(::File.join(node['apache']['dir'], 'mods-enabled',
      'proxy.load')) || node['tomcat']['apache_tomcat_bind_mod'] !~ /\Aproxy/
  end
end

execute 'enable module for apache-tomcat binding' do
  command "/usr/sbin/a2enmod #{node['tomcat']['apache_tomcat_bind_mod']}"
  not_if { ::File.symlink?(::File.join(node['apache']['dir'], 'mods-enabled',
    "#{$(node['tomcat']['apache_tomcat_bind_mod']).load})) }
end

include_recipe 'apache2::service'

template 'tomcat thru apache binding' do
  path ::File.join(node['apache']['dir'], 'conf.d', node['tomcat']['apache_tomcat_bind_config'])
  source 'apache_tomcat_bind.conf.erb'
  owner 'root'
  group 'root'
  mode 0644
  backup false
  notifies :restart, resources(:service => 'apache2')
end
```

To enable `mod_proxy`, you must enable the `proxy` module and a protocol-based module. You have two options for the protocol module:

- `HTTP: proxy_http`
- `Apache JServ Protocol (AJP): proxy_ajp`

AJP is an internal Tomcat protocol.

Both the recipe's [execute resources](#) run the `a2enmod` command, which enables the specified module by creating the required symlinks:

- The first `execute` resource enables the `proxy` module.
- The second `execute` resource enables the `protocol` module, which is set to `proxy_http` by default.

If you would rather use AJP, you can use custom JSON to override the `apache_tomcat_bind_mod` attribute and set it to `proxy_ajp`.

The `apache2::service` recipe is an AWS OpsWorks built-in recipe that defines the Apache service. For more information, see the [recipe](#) in the AWS OpsWorks GitHub repository.

The `template` resource uses `apache_tomcat_bind.conf.erb` to create a configuration file, named `tomcat_bind.conf` by default. It places the file in the `['apache']['dir']/.conf.d` directory. The `['apache']['dir']` attribute is defined in the built-in `apache2` attributes file, and is set by default to `/etc/httpd` (Amazon Linux), or `/etc/apache2` (Ubuntu). If the `template` resource creates or changes the configuration file, the `notifies` command schedules an Apache service restart.

```
<% if node['tomcat']['apache_tomcat_bind_mod'] == 'proxy_ajp' -%>
ProxyPass <%= node['tomcat']['apache_tomcat_bind_path'] %> ajp://localhost:<=%
node['tomcat']['ajp_port'] %>
ProxyPassReverse <%= node['tomcat']['apache_tomcat_bind_path'] %> ajp://local
host:<%= node['tomcat']['ajp_port'] %>
<% else %>
ProxyPass <%= node['tomcat']['apache_tomcat_bind_path'] %> http://localhost:<=%
node['tomcat']['port'] %>
ProxyPassReverse <%= node['tomcat']['apache_tomcat_bind_path'] %> http://local
host:<%= node['tomcat']['port'] %>
<% end -%>
```

The template uses the `ProxyPass` and `ProxyPassReverse` directives to configure the port used to pass traffic between Apache and Tomcat. Because both servers are on the same instance, they can use a `localhost` URL and are both set by default to `http://localhost:8080`.

## Configure Recipes

Configure recipes are assigned to the layer's Configure [lifecycle \(p. 176\)](#) event, which occurs on all of the stack's instances whenever an instance enters or leaves the online state. You use Configure recipes to adjust an instance's configuration to respond to the change, as appropriate. When you implement a Configure recipe, keep in mind that a stack configuration change might involve instances that have nothing to do with this layer. The recipe must be able to respond appropriately, which might mean doing nothing in some cases.

### tomcat::configure

The `tomcat::configure` recipe is intended for a layer's Configure lifecycle event.

```
include_recipe 'tomcat::context'
# Optional: Trigger a Tomcat restart in case of a configure event, if relevant
# settings in custom JSON have changed (e.g. java_opts/JAVA_OPTS):
#include_recipe 'tomcat::container_config'
```

The `tomcat::configure` recipe is basically a metarecipe that runs two dependent recipes.

1. The `tomcat::context` recipe create a web app context configuration file.

This file configures the JDBC resources that applications use to communicate with the MySQL instance, as discussed in the next section. Running this recipe in response to a configure event allows the layer to update the web app context configuration file if the database layer has changed.

2. The `tomcat::container_config` Setup recipe is run again to capture any changes in the container configuration.

The `include` for `tomcat::container_config` is commented out for this example. If you want to use custom JSON to modify Tomcat settings, you can remove the comment. A Configure lifecycle event then runs `tomcat::container_config`, which updates the Tomcat related configuration files, as described in [tomcat::container\\_config \(p. 246\)](#) and restarts the Tomcat service.

## tomcat::context

The Tomcat cookbook enables applications to access a MySQL database server, which can be running on a separate instance, by using a [J2EE DataSource](#) object. With Tomcat, you can enable the connection by creating and installing a web app context configuration file for each application. This file defines the relationship between the application and the JDBC resource that the application will use to communicate with the database. For more information, see [The Context Container](#).

The `tomcat::context` recipe's primary purpose is to create this configuration file.

```
include_recipe 'tomcat::service'

node[:deploy].each do |application, deploy|
  context_name = deploy[:document_root].blank? ? application : deploy[:document_root]

  template "context file for #{application} (context name: #{context_name})" do
    path ::File.join(node['tomcat']['catalina_base_dir'], 'Catalina', 'localhost', "#{context_name}.xml")
    source 'webapp_context.xml.erb'
    owner node['tomcat']['user']
    group node['tomcat']['group']
    mode 0640
    backup false
    only_if { node['datasources'][context_name] }
    variables(:resource_name => node['datasources'][context_name], :webapp_name => application)
    notifies :restart, resources(:service => 'tomcat')
  end
end
```

In addition to Tomcat cookbook attributes, this recipe uses the [stack configuration and deployment JSON \(p. 262\)](#) that AWS OpsWorks installs with the Configure event. AWS OpsWorks is based on Chef Solo and does not support data bags or search. Instead, the AWS OpsWorks service creates a JSON object that contains the information that recipes would typically obtain by using data bags or search and installs the object on each instance. The JSON object contains detailed information about the stack configuration, deployed apps, and any custom data that a user wants to include. Recipes can obtain data from stack configuration and deployment JSON by using standard Chef node syntax. For more information, see [Stack Configuration and Deployment JSON \(p. 262\)](#).

"Deployment JSON" refers to the `[:deploy]` namespace, which contains deployment-related attributes that are defined through the console or API, or generated by the AWS OpsWorks service. The deployment JSON includes an attribute for each deployed app, named with the app's short name. Each app attribute contains a set of attributes that characterize the app, such as the document root (`[:deploy][:app-name][:document_root]`). For example, here is a highly abbreviated version of the stack configuration and deployment JSON for an app named `my_1st_jsp` that was deployed to a Tomcat-based custom layer from an Amazon S3 archive.

```
{
  ...
  "datasources": {
    "ROOT": "jdbc/mydb"
  }
  ...
  "deploy": {
    "my_1st_jsp": {
      "document_root": "ROOT",
      ...
      "scm": {
        "password": null,
        "repository": "https://s3.amazonaws.com/.../my_1st_jsp.zip",
        "ssh_key": null,
        "scm_type": "archive",
        "user": null,
        "revision": null
      },
      ...
      "deploy_to": "/srv/www/my_1st_jsp",
      ...
      "database": {
        "password": "86la64jagj",
        "username": "root",
        "reconnect": true,
        "database": "my_1st_jsp",
        "host": "10.254.3.69"
      },
      ...
    }
  },
  ...
}
```

The `context` recipe first ensures that the service is defined for this Chef run by calling [tomcat::service \(p. 245\)](#). It then defines a `context_name` variable which represents the configuration file's name, excluding the `.xml` extension. If you use the default document root, `context_name` is set to the app's short name. Otherwise, it is set to the specified document root. The example discussed in [Create a Stack and Run an Application \(p. 256\)](#) sets the document root to "ROOT", so the context is ROOT and the configuration file is named `ROOT.xml`.

The bulk of the recipe goes through the list of deployed apps and for each app, uses the `webapp_context.xml.erb` template to create a context configuration file. The example deploys only one app, but the structure of the JSON requires you to treat it as a list of apps regardless.

The `webapp_context.xml.erb` template is not operating-system specific, so it is located in the `templates` directory's default subdirectory.

The recipe creates the configuration file as follows:

- Using default attribute values, the configuration file name is set to `context_name.xml` and installed in the `/etc/tomcat6/Catalina/localhost/` directory.

The `[ 'datasources' ]` node from the deployment JSON contains one or more attributes, each of which maps a context name to the JDBC data resource that the associated application will use to communicate with the database. The node and its contents are defined with custom JSON when you create the stack, as described later in [Create a Stack and Run an Application \(p. 256\)](#). The example has a single attribute that associates the ROOT context name with a JDBC resource named `jdbc/mydb`.

- Using default attribute values, the file's user and group are both set to the values defined by the Tomcat package: `tomcat` (Amazon Linux) or `tomcat6` (Ubuntu).
- The `template` resource creates the configuration file only if the `[ 'datasources' ]` node exists and includes a `context_name` attribute.
- The `template` resource defines two variables, `resource_name` and `webapp_name`.

`resource_name` is set to the resource name that is associated with `context_name` and `webapp_name` is set to the app's short name.

- The `template` resource restarts the Tomcat service to load and activate the changes.

The `webapp_context.xml.erb` template consists of a `Context` element that contains a `Resource` element with its own set of attributes.

```
<Context>
  <Resource name="<%= @resource_name %>" auth="Container"
type="javax.sql.DataSource"
  maxActive="20" maxIdle="5" maxWait="10000"
  username="<%= node['deploy'][@webapp_name]['database']['username'] %>"
  password="<%= node['deploy'][@webapp_name]['database']['password'] %>"
  driverClassName="com.mysql.jdbc.Driver"
  url="jdbc:mysql://<%= node['deploy'][@webapp_name]['data
base']['host'] %>:3306/<%= node['deploy'][@webapp_name]['database']['database']
%>"
  factory="org.apache.commons.dbcp.BasicDataSourceFactory" />
</Context>
```

These `Resource` attributes characterize the context configuration:

- **name**—The JDBC resource name, which is set to the `resource_name` value defined in `tomcat::context`.

For the example, the resource name is set to `jdbc/mydb`.

- **auth** and **type**—These are standard settings for JDBC `DataSource` connections.
- **maxActive**, **maxIdle**, and **maxWait**—The connection's active, idle, and wait times.
- **username**, and **password**—The database's user name and root password, which are obtained from the deployment JSON.
- **driverClassName**—The JDBC driver's class name, which is set to the MySQL driver.
- **url**—The connection URL.

The prefix depends on the database. It should be set to `jdbc:mysql` for MySQL, `jdbc:postgresql` for Postgres, and `jdbc:sqlserver` for SQL Server. The example sets the URL to `jd
bc:mysql://host_IP_Address:3306:simplejsp`, where `simplejsp` is the app's short name.

- **factory**—The `DataSource` factory, which is required for MySQL databases.

For more information on this configuration file, see the Tomcat wiki's [Using DataSources](#) topic.

## Deploy Recipes

Deploy recipes are assigned to the layer's Deploy [lifecycle \(p. 176\)](#) event. It typically occurs on all of the stack's instances whenever you deploy an app, although you can optionally restrict the event to only specified instances. AWS OpsWorks also runs the Deploy recipes on new instances, after the Setup recipes complete. The primary purpose of Deploy recipes is to deploy code and related files from a repository to the application server layer's instances. However, you often run Deploy recipes on other layers as well. This allows those layers' instances, for example, to update their configuration to accommodate the newly deployed app. When you implement a Deploy recipe, keep in mind a Deploy event does not necessarily mean that apps are being deployed to the instance. It could simply be a notification that apps are being deployed to other instances in the stack, to allow the instance to make any necessary updates. The recipe must be able to respond appropriately, which might mean doing nothing.

AWS OpsWorks automatically deploys apps of the standard app types to the corresponding built-in application server layers. To deploy apps to a custom layer, you must implement custom Deploy recipes that download the app's files from a repository to the appropriate location on the instance. However, you can often limit the amount of code you must write by using the built-in [deploy cookbook](#) to handle some aspects of deployment. For example, if you store your files in one of the supported repositories, the built-in cookbook can handle the details of downloading the files from the repository to the layer's instances.

The `tomcat::deploy` recipe is intended to be assigned to the Deploy lifecycle event.

```
include_recipe 'deploy'

node[:deploy].each do |application, deploy|
  opsworks_deploy_dir do
    user deploy[:user]
    group deploy[:group]
    path deploy[:deploy_to]
  end

  opsworks_deploy do
    deploy_data deploy
    app application
  end
  ...

```

The `tomcat::deploy` recipe uses the built-in `deploy` cookbook for aspects of deployment that aren't application specific. The `deploy` recipe (which is shorthand for the built-in `deploy::default` recipe) is a built-in recipe that handles the details of setting up the users, groups, and so on, based on data from the deployment JSON.

The recipe uses two built-in Chef definitions, `opsworks_deploy_dir` and `opworks_deploy` to install the application.

The `opsworks_deploy_dir` definition sets up the directory structure, based on data from the app's deployment JSON. Definitions are basically a convenient way to package resource definitions, and are located in a cookbook's `definitions` directory. Recipes can use definitions much like resources, but the definition itself does not have an associated provider, just the resources that are included in the definition. You can define variables in the recipe, which are passed to the underlying resource definitions. The `tomcat::deploy` recipe sets `user`, `group`, and `path` variables based on data from the deployment JSON. They are passed to the definition's [directory resource](#), which manages the directories.

### Note

Your deployed app's user and group are determined by the `[:opsworks][:deploy_user][:user]` and `[:opsworks][:deploy_user][:group]` attributes, which are

defined in the [built-in deploy cookbook's deploy.rb attributes file](#). The default value of `[:opsworks][:deploy_user][:user]` is `deploy`. The default value of `[:opsworks][:deploy_user][:group]` depends on the instance's operating system:

- For Ubuntu instances, the default group is `www-data`.
- For Amazon Linux instances that are members of a Rails App Server layer that uses Nginx and Unicorn, the default group is `nginx`.
- For all other Amazon Linux instances, the default group is `apache`.

You can change either setting by using custom JSON or a custom attributes file to override the appropriate attribute. For more information, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

The other definition, `opsworks_deploy`, handles the details of checking out the app's code and related files from the repository and deploying them to the instance, based on data from the deployment JSON. You can use this definition for any app type; deployment details such as the directory names are specified in the console or through the API and put in the deployment JSON. However, `opsworks_deploy` works only for the four [supported repository types \(p. 154\)](#): Git, Subversion, S3, and HTTP. You must implement this code yourself if you want to use a different repository type.

You install an app's files in the Tomcat `webapps` directory. A typical practice is to copy the files directly to `webapps`. However, AWS OpsWorks deployment is designed to retain up to five versions of an app on an instance, so you can roll back to an earlier version if necessary. AWS OpsWorks therefore does the following:

1. Deploys apps to a distinct directory whose name contains a time stamp, such as `/srv/www/my_1st_jsp/releases/20130731141527`.
2. Creates a symlink named `current`, such as `/srv/www/my_1st_jsp/current`, to this unique directory.
3. If does not already exist, creates a symlink from the `webapps` directory to the `current` symlink created in Step 2.

If you need to roll back to an earlier version, modify the `current` symlink to point to a distinct directory containing the appropriate timestamp, for example, by changing the link target of `/srv/www/my_1st_jsp/current`.

The middle section of `tomcat::deploy` sets up the symlink.

```
...
current_dir = ::File.join(deploy[:deploy_to], 'current')
webapp_dir = ::File.join(node['tomcat']['webapps_base_dir'], deploy[:document_root].blank? ? application : deploy[:document_root])

# opsworks_deploy creates some stub dirs, which are not needed for typical
# webapps
ruby_block "remove unnecessary directory entries in #{current_dir}" do
  block do
    node['tomcat']['webapps_dir_entries_to_delete'].each do |dir_entry|
      ::FileUtils.rm_rf(::File.join(current_dir, dir_entry), :secure => true)
    end
  end
end
end
```

```
link webapp_dir do
  to current_dir
  action :create
end
...
```

The recipe first creates two variables, `current_dir` and `webapp_dir` to represent the `current` and `webapp` directories, respectively. It then uses a `link` resource to link `webapp_dir` to `current_dir`. The AWS OpsWorks `deploy::default` recipe creates some stub directories that aren't required for this example, so the middle part of the excerpt removes them.

The final part of `tomcat::deploy` restarts the Tomcat service, if necessary.

```
...
include_recipe 'tomcat::service'

execute 'trigger tomcat service restart' do
  command '/bin/true'
  not_if { node['tomcat']['auto_deploy'].to_s == 'true' }
  notifies :restart, resources(:service => 'tomcat')
end
end

include_recipe 'tomcat::context'
```

The recipe first runs `tomcat::service`, to ensure that the service is defined for this Chef run. It then uses an `execute` resource to notify the service to restart, but only if `['tomcat']['auto_deploy']` is set to `'true'`. Otherwise, Tomcat listens for changes in its `webapps` directory, which makes an explicit Tomcat service restart unnecessary.

**Note**

The `execute` resource doesn't actually execute anything substantive; `/bin/true` is a dummy shell script that simply returns a success code. It is used here simply as a convenient way to generate a restart notification. As mentioned earlier, using notifications ensures that services are not restarted too frequently.

Finally, `tomcat::deploy` runs `tomcat::context`, which updates the web app context configuration file if you have changed the back end database.

## Create a Stack and Run an Application

This section shows how to use the Tomcat cookbook to implement a basic stack setup that runs a simple Java server pages (JSP) application named SimpleJSP. The stack consists of a Tomcat-based custom layer named TomCustom and a MySQL layer. SimpleJSP is deployed to TomCustom and displays some information from the MySQL database. If you are not already familiar with the basics of how to use AWS OpsWorks, you should first read [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#).

### The SimpleJSP Application

The SimpleJSP application demonstrates the basics of how to set up a database connection and retrieve data from the stack's MySQL database.

```
<html>
```

```

<head>
    <title>DB Access</title>
</head>
<body>
<%@ page language="java" import="java.sql.* , javax.naming.* , javax.sql.*" %>

<%
    StringBuffer output = new StringBuffer();
    DataSource ds = null;
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        Context initCtx = new InitialContext();
        ds = (DataSource) initCtx.lookup("java:comp/env/jdbc/mydb");
        con = ds.getConnection();
        output.append("Databases found:<br>");
        stmt = con.createStatement();
        rs = stmt.executeQuery("show databases");
        while (rs.next()) {
            output.append(rs.getString(1));
            output.append("<br>");
        }
    }
    catch (Exception e) {
        output.append("Exception: ");
        output.append(e.getMessage());
        output.append("<br>");
    }
    finally {
        try {
            if (rs != null) {
                rs.close();
            }
            if (stmt != null) {
                stmt.close();
            }
            if (con != null) {
                con.close();
            }
        }
        catch (Exception e) {
            output.append("Exception (during close of connection): ");
            output.append(e.getMessage());
            output.append("<br>");
        }
    }
%>
<%= output.toString() %>
</body>
</html>

```

SimpleJSP uses a `DataSource` object to communicate with the MySQL database. Tomcat uses the data in the [web app context configuration file \(p. 251\)](#) to create and initialize a `DataSource` object and bind it to a logical name. It then registers the logical name with a Java Naming and Directory Interface (JNDI) naming service. To get an instance of the appropriate `DataSource` object, you create an `InitialContext` object and pass the resource's logical name to the object's `lookup` method, which retrieves the appropriate

object. The SimpleJSP example's logical name, `java:comp/env/jdbc/mydb`, has the following components:

- The root namespace, `java`, which is separated from the rest of the name by a colon (`:`).
- Any additional namespaces, separated by forward slashes (`/`).  
Tomcat automatically adds resources to the `comp/env` namespace.
- The resource name, which is defined in the web app context configuration file and separated from the namespaces by a forward slash.

The resource name for this example is `jdbc/mydb`.

To establish a connection to the database, SimpleJSP does the following:

1. Calls the `DataSource` object's `getConnection` method, which returns a `Connection` object.
2. Calls the `Connection` object's `createStatement` method to create a `Statement` object, which you use to communicate with the database.
3. Communicates with the database by calling the appropriate `Statement` method.

SimpleJSP calls `executeQuery` to execute a `SHOW DATABASES` query, which lists the server's databases.

The `executeQuery` method returns a `ResultSet` object, which contains the query results. SimpleJSP gets the database names from the returned `ResultSet` object and concatenates them to create an output string. Finally, the example closes the `ResultSet`, `Statement`, and `Connection` objects. For more information about JSP and JDBC, see [JavaServer Pages Technology](#) and [JDBC Basics](#), respectively.

To use SimpleJSP with a stack, you must put it in a repository. You can use any of the supported repositories, but to use SimpleJSP with the example stack discussed in the following section, you must put it in a public S3 archive. For information on how to use the other standard repositories, see [Cookbook Repositories \(p. 154\)](#).

### To put SimpleJSP in an S3 archive repository

1. Copy the example code to a file named `simplejsp.jsp` and put the file in a directory named `simplejsp`.
2. Create a `.zip` archive of the `simplejsp` directory.
3. Create a public Amazon S3 bucket, upload `simplejsp.zip` to the bucket, and make the file public.

For a description of how to perform this task, see [Get Started With Amazon Simple Storage Service](#).

## Create a Stack

To run SimpleJSP you need a stack with the following layers.

- A MySQL layer, that supports the back end MySQL server.
- A custom layer that uses the Tomcat cookbook to support Tomcat server instances.

### To create the stack

1. On the AWS OpsWorks dashboard, click **Add Stack** to create a new stack and click **Advanced >** to display all options. Configure the stack as follows.

For the remaining options, you can accept the defaults.

- **Name**—A user-defined stack name; this example uses TomStack.
- **Use custom Chef cookbooks**—Set the toggle to **Yes**, which displays some additional options.
- **Repository type**—Git.
- **Repository URL**—`git://github.com/amazonwebservices/opsworks-example-cookbooks.git`.
- **Custom Chef JSON**—Add the following JSON:

```
{
  "tomcat": {
    "base_version": 7,
    "java_opts": "-Djava.awt.headless=true -Xmx256m"
  },
  "datasources": {
    "ROOT": "jdbc/mydb"
  }
}
```

The custom JSON does the following:

- Overrides the Tomcat cookbook's [ 'base\_version' ] attribute to set the Tomcat version to 7; the default value is 6.
- Overrides the Tomcat cookbook's [ 'java\_opts' ] attribute to specify that the instance is headless and set the JVM maximum heap size to 256MB; the default value sets no options for instances running Amazon Linux.
- Specifies the [ 'datasources' ] attribute value, which assigns a JDBC resource name (jdbc/mydb) to the web app context name (ROOT), as discussed in [tomcat::context \(p. 251\)](#).

This last attribute has no default value; you must set it with custom JSON.

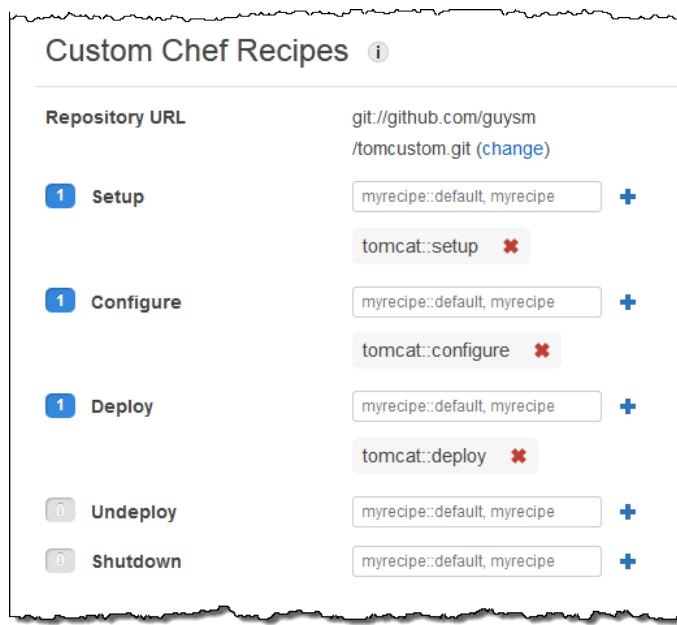


2. Click **Add a layer**. For **Layer type**, select **MySQL**. Then click **Add Layer**.
3. Click **Instances** in the navigation pane and then click **Add an instance**. Click **Add Instance** to accept the defaults. On the line for the instance, click **start**.
4. Return to the **Layers** page and click **+ Layer** to add a layer. For **Layer type**, click **Custom**. The example uses `TomCustom` and `tomcustom` as the layer's name and short name, respectively.

## Add Layer

Layer type	<input style="border: 1px solid #ccc; padding: 2px 10px;" type="button" value="Custom"/>
<p>The Custom layer allows you to create a fully customized layer. Standard recipes handle basic setup and configuration for the layer instances, and you implement custom Chef recipes to install and configure any required software. You can create as many custom layers as you require. <a href="#">Learn more</a>.</p>	
Name	<input type="text" value="TomCustom"/>
Short name	<input type="text" value="tomcustom"/>

5. On the **Layers** page, for the custom layer, click **Recipes** and then click **Edit**. Under **Custom Chef Recipes**, assign Tomcat cookbook recipes to the layer's lifecycle events, as follows:
  - For **Setup**, type `tomcat::setup` and click **+**.
  - For **Configure**, type `tomcat::configure` and click **+**.
  - For **Deploy**, type `tomcat::deploy` and click **+**. Then click **Save**.



6. Click **Apps** in the navigation pane and then click **Add an app**. Specify the following options and then click **Add App**:

Use default settings for the other options.

- **Name**—The app's name; the example uses SimpleJSP and the short name generated by AWS OpsWorks will be simplejsp.
- **App type**—Set this option to **Other**.

AWS OpsWorks automatically deploys standard app types to the associated server instances. If you set **App type** to other, AWS OpsWorks simply runs the Deploy recipes, and lets them handle deployment.

- **Document root**—Set this option to `ROOT`.

The **Document root** value specifies the context name.

- **Repository type**—Set this option to **Http Archive**, which is the appropriate setting for public S3 archives.

The options also include an **S3 Archive** repository type, but you only use this for private S3 archives.

- **Repository URL**—Set this to the app's S3 URL that you created earlier.

Settings	
Name	SimpleJSP
App type	Other
Document root	ROOT

Application Source	
Repository type	Http Archive
Repository URL	vs.com/opsworks-tomcat/simplejsp.zip
User name	Optional
Password	Optional

Add Domains...

7. Use the **Instances** page to add an instance to the TomCustom layer and start it. AWS OpsWorks automatically runs the Deploy recipes on a new instance after the Setup recipes complete, so starting the instance also deploys SimpleJSP.
8. When the TomCustom instance is online, click the instance name on the **Instances** page to see its details. Copy the public IP address. Then construct a URL as follows: `http://publicIP/tc/app-name.jsp`. For the example, this URL will look something like `http://50.218.191.172/tc/simplejsp.jsp`.

#### Note

The Apache URL that forwards requests to Tomcat is set to the default `['tomcat'][['apache_tomcat_bind_path']]` attribute, `/tc/`. The SimpleJSP document root is set to `ROOT` which is a special value that resolves to `/`. The URL is therefore `../tc/simplejsp.jsp`.

9. Paste the URL from the previous step into your browser. You should see the following:

```
Databases found:
information_schema
simplejsp
test
```

**Note**

If your stack has a MySQL instance, AWS OpsWorks automatically creates a database for each app, named with the app's short name.

# Stack Configuration and Deployment JSON

When AWS OpsWorks runs a command on an instance—for example, a deploy command in response to a Deploy lifecycle event—it sends a JSON object to the instance that describes the stack's current configuration and. For Deploy events and [Execute Recipes stack commands \(p. 52\)](#), the object includes some additional deployment information. The object's attributes are incorporated into the instance's node object, as described in [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#). You can also view the JSON directly by using the agent CLI. For more information, see [How to Obtain a Stack Configuration JSON \(p. 267\)](#). For a list of commonly used stack configuration and deployment attributes, including fully qualified node names, see [Appendix C: AWS OpsWorks Attribute Reference \(p. 378\)](#).

The following sections show the JSON associated with a Configure event and a Deploy event for a simple stack, which consists of the following:

- A PHP App Server layer with two instances.
- An HAProxy layer with one instance

The JSON examples are from one of the PHP App Server instances, **php-app1**.

**Note**

Instances and layers are identified by their short names in the configuration JSON. For example, the PHP App Server layer is identified by "php-app".

## Configure JSON

The following example is the JSON object for a Configure event, which occurs on every instance in the stack when an instance comes online or goes offline. The JSON for a Configure event consists of the built-in stack configuration JSON and any custom stack JSON that was specified prior to the event (none in this example). It has been edited for length. For a detailed description of the various attributes, see [Appendix C: AWS OpsWorks Attribute Reference \(p. 378\)](#).

```
{  
    "opsworks": {  
        "layers": {  
            "php-app": {  
                "id": "4a2a56c8-f909-4b39-81f8-556536d20648",  
                "instances": {  
                    "php-app2": {  
                        "elastic_ip": null,  
                        "region": "us-west-2",  
                        "booted_at": "2013-02-26T20:41:10+00:00",  
                        "ip": "192.112.235.192",  
                        "aws_instance_id": "i-34037f06",  
                        "availability_zone": "us-west-2a",  
                        "instance_type": "c1.medium",  
                        "private_dns_name": "ip-10-252-0-203.us-west-2.compute.internal",  
                        "private_ip": "10.252.0.203",  
                        "created_at": "2013-02-26T20:39:39+00:00",  
                        "status": "online",  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        "backends": 8,
        "public_dns_name": "ec2-192-112-235-192.us-west-2.compute.amazonaws.com"
    },
    "php-app1": {
        ...
    },
    "name": "PHP Application Server"
},
"lb": {
    "id": "15c86142-d836-4191-860f-f4d310440f14",
    "instances": {
        "lb1": {
            ...
        }
    },
    "name": "Load Balancer"
}
},
"agent_version": "104",
"applications": [
],
"stack": {
    "name": "MyStack"
},
"ruby_version": "1.8.7",
"sent_at": 1361911623,
"ruby_stack": "ruby_enterprise",
"instance": {
    "layers": [
        "php-app"
    ],
    "region": "us-west-2",
    "ip": "192.112.44.160",
    "id": "45ef378d-b87c-42be-alb9-b67c48edaf4",
    "aws_instance_id": "i-32037f00",
    "availability_zone": "us-west-2a",
    "private_dns_name": "ip-10-252-84-253.us-west-2.compute.internal",
    "instance_type": "c1.medium",
    "hostname": "php-app1",
    "private_ip": "10.252.84.253",
    "backends": 8,
    "architecture": "i386",
    "public_dns_name": "ec2-192-112-44-160.us-west-2.compute.amazonaws.com"
},
"activity": "configure",
"rails_stack": {
    "name": null
},
"deployment": null,
"valid_client_activities": [
    "reboot",
    "stop",
    "setup",
    "configure",
    "update_dependencies",
```

```

        "install_dependencies",
        "update_custom_cookbooks",
        "execute_recipes"
    ],
},
"opsworks_custom_cookbooks": {
    "recipes": [
        ],
        "enabled": false
    },
    "recipes": [
        "opsworks_custom_cookbooks::load",
        "opsworks_ganglia::configure-client",
        "ssh_users",
        "agent_version",
        "mod_php5_apache2::php",
        "php::configure",
        "opsworks_stack_state_sync",
        "opsworks_custom_cookbooks::execute",
        "test_suite",
        "opsworks_cleanup"
    ],
    "opsworks_rubygems": {
        "version": "1.8.24"
    },
    "ssh_users": {
    },
    "opsworks_bundler": {
        "manage_package": null,
        "version": "1.0.10"
    },
    "deploy": {
    }
}
}

```

Most of the information is in the `opsworks` attribute, which is often referred to as a namespace. The following list describes the key attributes:

- **layers attributes** – A set of attributes, each of which describes the configuration of one of the stack's layers. The layers are identified by their shortnames, `php-app` and `lb` for this example. For more information about layer shortnames for other layers, see [Appendix A: AWS OpsWorks Layer Reference \(p. 354\)](#).

Every layer has an `instances` element, each of whose attributes describes one of the layers' online instances and bears the instance's short name. The PHP App Server layer has two instances, `php-app1` and `php-app2`. The HAProxy layer has one instance, `lb1`. Each instance attribute contains a set of attributes that specify values such as the instance's private IP address and private DNS name. For brevity, the example shows only `php-app2` in detail; the others contain similar information.

#### Note

The `instances` element contains only those instances that are in the online state when the particular stack and deployment JSON is created.

- `applications` – A list of deployed apps, not found in this example.
- `stack` – The stack name; `MyStack` in this example.

- `instance` – The instance that this JSON resides on; `php-app1` in this example. Recipes can use this attribute to obtain information about the instance that they are running on, such as the instance's public IP address.
- `activity` – The activity that produced the JSON; a `configure` event in this example.
- `rails_stack` – The Rails stack for stacks that include a Rails App Server layer.
- `deployment` – Whether this JSON is associated with a deployment. It is set to `null` for this example because the JSON is associated with a Configure event.
- `valid_client_activities` – A list of valid client activities.

The `opsworks` attribute is followed by several top-level attributes, including the following:

- `opsworks_custom_cookbooks` – Whether custom cookbooks are enabled. If so, the attribute includes a list of custom recipes.
- `recipes` – The recipes that were run by this activity.
- `opsworks_rubygems` – The instance's RubyGems version.
- `ssh_users` – Lists SSH users; none in this example.
- `opsworks_bundler` – The bundler version and whether it is enabled.
- `deploy` – Information about deployment activities; none in this example.

## Deploy JSON

The JSON for a Deploy event or [Execute Recipes stack command \(p. 52\)](#) consists of the built-in stack configuration and deployment JSON, and any custom stack or deployment JSON (none for this example). The following example is a JSON object from `php-app1` that is associated with the Deploy event that deployed the SimplePHP app to the stack's PHP instances. Much of the object is stack configuration JSON that is similar to the JSON for the Configure event described in the previous section, so the example focuses primarily on the deployment-specific parts. For a detailed description of the various attributes, see [Appendix C: AWS OpsWorks Attribute Reference \(p. 378\)](#).

```
{  
    ...  
    "opsworks": {  
        ...  
        "activity": "deploy",  
        "applications": [  
            {  
                "slug_name": "simplephp",  
                "name": "SimplePHP",  
                "application_type": "php"  
            }  
        ],  
        "deployment": "5e6242d7-8111-40ee-bddb-00de064ab18f",  
        ...  
    },  
    ...  
    {  
        "ssh_users": {  
        },  
        "deploy": {  
            "simplephpapp": {  
                "application": "simplephpapp",  
                "application_type": "php",  
                ...  
            }  
        }  
    }  
}
```

```

"environment": {
    "USER_ID": "168424",
    "USER_KEY": "somepassword"
},
"auto_bundle_on_deploy": true,
"deploy_to": "/srv/www/simplephpapp",
"deploying_user": "arn:aws:iam::645732743964:user/guysm",
"document_root": null,
"domains": [
    "simplephpapp"
],
"migrate": false,
"mounted_at": null,
"rails_env": null,
"restart_command": "echo 'restarting app'",
"sleep_before_restart": 0,
"ssl_support": false,
"ssl_certificate": null,
"ssl_certificate_key": null,
"ssl_certificate_ca": null,
"scm": {
    "scm_type": "git",
    "repository": "git://github.com/amazonwebservices/opsworks-demo-php-simple-app.git",
    "revision": "version1",
    "ssh_key": null,
    "user": null,
    "password": null
},
"symlink_before_migrate": {
    "config/opsworks.php": "opsworks.php"
},
"symlinks": {
},
"database": {
},
"memcached": {
    "host": null,
    "port": 11211
},
"stack": {
    "needs_reload": false
}
}
},
}
}

```

The `opsworks` attribute is largely identical to the `configure` JSON example in the previous section. The following sections are most relevant to app deployment:

- `activity` – The event that is associated with the JSON; a `Deploy` event in this example.
- `applications` – A list of elements, one for each app, that provide the apps' names, slug names, and types. The slug name is a short name that AWS OpsWorks generates from the app name in order to identify the app. The slug name for SimplePHP is `simplephp`.
- `deployment` – The deployment ID, which uniquely identifies the deployment.

The deploy section provides information about the apps that are currently being deployed. It is used, for example, by the built-in Deploy recipes. The section has one attribute for each deployed app. Each app attribute includes the following information:

- `environment` – Contains any environment variables that you have defined for the app. For more information, see [Environment Variables \(p. 131\)](#).
- `domains` – By default, the domain is the app's slug name, which is the case in this example. If you have assigned custom domains, they appear here as well. For more information, see [Using Custom Domains \(p. 146\)](#).
- `application` – The app's slug name.
- `scm` – This element contains the information required to download the app's files from its repository; a Git repository in this example.
- `database` – Database information, if the stack includes a MySQL layer.
- `document_root` – The document root, which is set to `null` in this example, indicating that the root is public.
- `ssl_certificate_ca`, `ssl_support`, `ssl_certificate_key` – Indicates whether the app has SSL support. If so, it includes the certificate information.
- `deploy_to` – The location of the app's files on the instance.

## Using Stack Configuration JSON Values in Custom Recipes

Your custom recipes can access stack configuration JSON values by constructing Chef nodes from the structure. Each level in the structure corresponds to a node element. The following example obtains the activity that is associated with the JSON:

```
node[ :opsworks][ :activity]
```

The following example obtains the private IP address of the HAProxy layer's first instance:

```
node[ :opsworks][ :layers][ :lb][ :instances].first[:private_ip]
```

For more information, see the documentation on [Opscode Chef](#).

## How to Obtain a Stack Configuration JSON

AWS OpsWorks stores recent stack configuration JSONs locally as files named `time-stamp.json`, where `time-stamp` is the date and time that the JSON was installed on the system. There are typically multiple JSON files in this folder. You can identify the appropriate file by examining the time stamps on the deployments page. The simplest way to retrieve a stack configuration JSON is to use SSH to connect to the instance and call the `get_json` agent CLI command to retrieve the JSON. For example, the following command retrieves the most recent stack configuration JSON.

```
sudo opsworks-agent-cli get_json
```

For more information about how to use SSH with AWS OpsWorks instances, see [Using SSH to Communicate with an Instance \(p. 119\)](#). For more information about how to use the agent CLI, see [Appendix B: Instance Agent CLI \(p. 368\)](#).

# Monitoring

---

You can monitor your stacks in the following ways.

- AWS OpsWorks uses Amazon CloudWatch to provide thirteen custom metrics with detailed monitoring for each instance in the stack.
- AWS OpsWorks integrates with AWS CloudTrail to log every AWS OpsWorks API call and store the data in an Amazon S3 bucket.
- You can use Amazon CloudWatch Logs to monitor your stack's system, application, and custom logs.

## Topics

- [Monitoring Stacks using Amazon CloudWatch \(p. 268\)](#)
- [Logging AWS OpsWorks API Calls By Using AWS CloudTrail \(p. 272\)](#)
- [Using Amazon CloudWatch Logs with AWS OpsWorks \(p. 275\)](#)

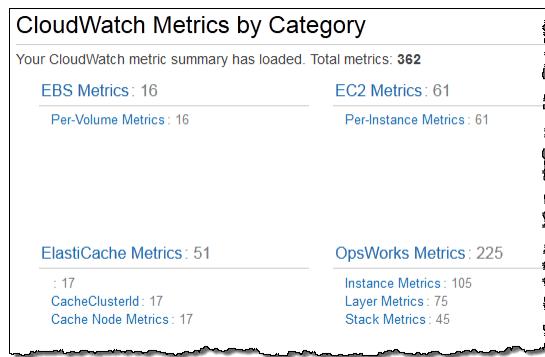
## Monitoring Stacks using Amazon CloudWatch

AWS OpsWorks uses Amazon CloudWatch to provide thirteen custom metrics with detailed monitoring for each instance in the stack and summarizes the data for your convenience on the **Monitoring** page. You can view metrics for the entire stack, a specified layer, or a specified instance. AWS OpsWorks metrics are distinct from Amazon EC2 metrics. If you want additional detailed metrics, you can enable them through the CloudWatch console, but they will typically require additional charges.

You can also view the underlying data on CloudWatch, as follows:

### To view OpsWorks metrics on CloudWatch

1. Open the Amazon CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. From the navigation bar, select **US East (Northern Virginia) Region**. AWS OpsWorks currently sends all CloudWatch metrics to **US East (Northern Virginia) Region**, regardless of the stack's region.
3. Click **Metrics** in the navigation pane.
4. Under OpsWorks Metrics:, click **Instance Metrics**, **Layer Metrics**, or **Stack Metrics**.



You can also view AWS OpsWorks metrics by clicking **OpsWorks** in the navigation pane.

**Note**

AWS OpsWorks collects metrics using a process running on each instance (the instance agent) and CloudWatch collects metrics using the hypervisor, so the values on the CloudWatch console might differ slightly from the corresponding values on the **Monitoring** page.

You can also use CloudWatch console to set alarms. For a list of CloudWatch metrics, see [AWS OpsWorks Dimensions and Metrics](#). For more information, see [Amazon CloudWatch](#).

**Topics**

- [Metrics \(p. 269\)](#)
- [Stack Metrics \(p. 270\)](#)
- [Layer Metrics \(p. 270\)](#)
- [Instance Metrics \(p. 271\)](#)

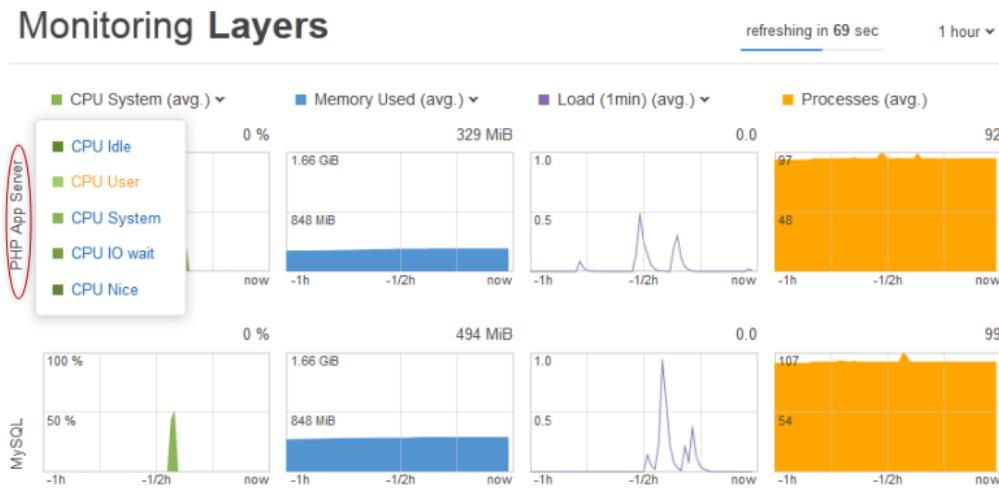
## Metrics

The **Monitoring** page displays graphs of the following custom metrics.

- **CPU** graph ranges from 0 - 100% and can display the following metrics:
  - **CPU Idle** – The percentage of time that the CPU is idle.
  - **CPU User** – The percentage of time that the CPU is handling user operations.
  - **CPU System** – The percentage of time that the CPU is handling system operations.
  - **CPU IO wait** – The percentage of time that the CPU is waiting for input/output operations.
  - **CPU Nice** – The percentage of time that the CPU is handling processes with a positive nice value, which have lower scheduling priority. For information, see [nice\(Unix\)](#).
- **Memory** graph ranges from 0 to the total amount of memory and can display the following metrics:
  - **Memory Total** – The total amount of memory.
  - **Memory Used** – The amount of memory in use.
  - **Memory SWAP** – The amount of swap space.
  - **Memory Free** – The amount of free memory.
  - **Memory Buffers** – The amount of buffered memory.
- **Load** graph ranges from 0 to the current maximum value and displays the following metrics:
  - **Load (1 min)** – The load averaged over a 1-minute window.
  - **Load (5 min)** – The load averaged over a 5-minute window.
  - **Load (15 min)** – The load averaged over a 15-minute window.
- **Processes** graph ranges from 0 to the current maximum value, and displays a single metric: the number of active processes.

## Stack Metrics

To view a summary of metrics for an entire stack, select a stack in the AWS OpsWorks **Dashboard** and then click **Monitoring** in the navigation pane. The following example is for a stack with a PHP App Server and MySQL layer.



The stack view displays graphs of the four types of metrics for each layer over a specified time period: 1 hour, 8 hours, 24 hours, 1 week, or 2 weeks. Note the following:

- AWS OpsWorks periodically updates the graphs; the countdown timer at the upper right indicates the time remaining until the next update,
- If a layer has more than one instance, the graphs display average values for the layer.
- You can specify the time period by clicking the list at the upper right and selecting your preferred value.

For each metric type, you can use the list at the top of the graph to select the particular metric that you want to view.

## Layer Metrics

To view metrics for a particular layer, click the layer name in the **Monitoring Layers** view. The following example shows metrics for the PHP App Server layer, which has two instances.



The metric types are the same as for the stack metrics, and for each type, you can use the list at the top of the graph to select the particular metric that you want to view.

**Tip**

You can also display layer metrics by going to the layer's details page and clicking **Monitoring** at the upper right.

## Instance Metrics

To view metrics for a particular instance, click the instance name in the layer monitoring view. The following example shows metrics for the PHP App Server layer's php-app1 instance.



The graphs summarize all the available metrics for each metric type. To get exact values for a particular point in time, use your mouse to move the slider (indicated by the red arrow in the previous illustration) to the appropriate position.

**Tip**

You can also display instance metrics by going to the instance's details page and clicking **Monitoring** at the upper right.

## Logging AWS OpsWorks API Calls By Using AWS CloudTrail

AWS OpsWorks is integrated with CloudTrail, a service that captures API calls made by or on behalf of AWS OpsWorks in your AWS account and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls from the AWS OpsWorks console or from the AWS OpsWorks API. Using the information collected by CloudTrail, you can determine what request was made to AWS OpsWorks, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

## AWS OpsWorks Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to AWS OpsWorks actions are tracked in log files. AWS OpsWorks records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

### Important

Although your stacks can be in any AWS region, all calls made by AWS OpsWorks on your behalf originate in the US East (Northern Virginia) Region. To log API calls made on your behalf to other regions, you must enable CloudTrail in those regions.

All of the AWS OpsWorks actions are logged and are documented in the [AWS OpsWorks API Reference](#). For example, each call to [CreateLayer](#), [DescribeInstances](#) or [StartInstance](#) generates a corresponding entry in the CloudTrail log files.

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the **userIdentity** field in the [CloudTrail Event Reference](#).

You can store your log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish Amazon SNS notifications when new log files are delivered if you want to take quick action upon log file delivery. For more information, see [Configuring Amazon SNS Notifications](#).

You can also aggregate AWS OpsWorks log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket. For more information, see [Aggregating CloudTrail Log Files to a Single Amazon S3 Bucket](#).

## Understanding AWS OpsWorks Log File Entries

CloudTrail log files can contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows a CloudTrail log entry that demonstrates the [CreateLayer](#) and [DescribeInstances](#) actions.

### Note

Potentially sensitive information, including database passwords and custom JSON, is redacted and does not appear in the CloudTrail logs. The information is instead represented by `**redacted**`.

```
{  
  "Records": [  
    {  
      "awsRegion": "us-east-1",  
      "eventID": "342cd1ec-8214-4a0f-a68f-8e6352feb5af",  
      "eventName": "CreateLayer",  
      "eventSource": "opsworks.amazonaws.com",  
      "eventTime": "2013-02-18T22:14:00Z",  
      "requestParameters": {  
        "layerName": "MyLayer",  
        "stackId": "12345678-1234-1234-1234-1234567890ab",  
        "versionLabel": "1.0",  
        "description": "A test layer",  
        "type": "Standard",  
        "autoMinorVersionUpgrade": false,  
        "tags": [{"Key": "MyTag", "Value": "MyValue"}]  
      },  
      "responseElements": {},  
      "userAgent": "awscli/1.11.100 Python/3.6.10 Darwin/19.6.0",  
      "version": "1"  
    }  
  ]  
}
```

```
"eventTime": "2014-05-28T16:05:29Z",
"eventVersion": "1.01"ed,
"requestID": "e3952a2b-e681-11e3-aa71-81092480ee2e",
"requestParameters": {
    "attributes": {},
    "customRecipes": {},
    "name": "2014-05-28 16:05:29 +0000 a073",
    "shortname": "customcf4571d5c0d6",
    "stackId": "a263312e-f937-4949-a91f-f32b6b641b2c",
    "type": "custom"
},
"responseElements": null,
"sourceIPAddress": "198.51.100.0",
"userAgent": "aws-sdk-ruby/1.20.0 ruby/1.9.3 x86_64-linux",
"userIdentity": {
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "accountId": "111122223333",
    "arn": "arn:aws:iam::111122223333:user/A-User-Name",
    "principalId": "AKIAI44QH8DHBEXAMPLE",
    "type": "IAMUser",
    "userName": "A-User-Name"
}
},
{
    "awsRegion": "us-east-1",
    "eventID": "a860d8f8-c1eb-449b-8f55-eafc373b49a4",
    "eventName": "DescribeInstances",
    "eventSource": "opsworks.amazonaws.com",
    "eventTime": "2014-05-28T16:05:31Z",
    "eventVersion": "1.01",
    "requestID": "e4691bfd-e681-11e3-aa71-81092480ee2e",
    "requestParameters": {
        "instanceIds": [
            "218289c4-0492-473d-a990-3fbe1efa25f6"
        ]
    },
    "responseElements": null,
    "sourceIPAddress": "198.51.100.0",
    "userAgent": "aws-sdk-ruby/1.20.0 ruby/1.9.3 x86_64-linux",
    "userIdentity": {
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "accountId": "111122223333",
        "arn": "arn:aws:iam::111122223333:user/A-User-Name",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "type": "IAMUser",
        "userName": "A-User-Name"
    }
}
]
```

# Using Amazon CloudWatch Logs with AWS OpsWorks

Every instance in your stack has a variety of log files, including system logs, application logs, and perhaps custom logs. You usually need to monitor at least some of these logs on a regular basis, for example, to detect when an application server is generating more than the expected rate of 404 status codes. To simplify the process of monitoring logs on multiple instances, AWS OpsWorks supports Amazon CloudWatch Logs.

With CloudWatch Logs, you install an agent on your instances that monitors selected logs for the occurrence of a user-specified pattern. For example, you can monitor logs for the occurrence of a literal term such as `NullReferenceException`, or count the number of such occurrences. The agent sends the logs to CloudWatch Logs, and you can use the CloudWatch Logs console or CLI to configure metrics to send you a notification when, for example, the number of errors in the log exceeds a specified threshold. For a general discussion of CloudWatch Logs, see [Monitoring System, Application, and Custom Log Files](#).

To enable CloudWatch Logs monitoring on an AWS OpsWorks stack, you must do the following.

- Update your instance profile so that the CloudWatch Logs agent has appropriate permissions.  
You typically need to do this task only once. You can then use the updated instance 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 CloudWatch Logs agent on each instance.

You can automate the last two items by creating 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 CloudWatch Logs. For a detailed walkthrough of how to use this approach to enable CloudWatch Logs on a simple AWS OpsWorks stack, see [Quick Start: Install the CloudWatch Logs Agent Using AWS OpsWorks and Chef](#).

# Security and Permissions

---

When you create an AWS account, you have a single sign-in that provides unlimited access to all AWS services and resources. An alternative approach is [AWS Identity and Access Management \(IAM\)](#), which enables you to securely control how each user accesses AWS services and resources. AWS OpsWorks integrates with AWS Identity and Access Management (IAM) to let you control the following:

- How individual users can interact with AWS OpsWorks.

For example, you can allow some users to deploy apps but not modify the stack while allowing other users full access but only to certain stacks, and so on.

- How AWS OpsWorks can act on your behalf to manage stack resources such as Amazon EC2 instances.

IAM provides a policy template with predefined permissions for this task.

- How apps that run on EC2 instances controlled by AWS OpsWorks can access other AWS resources such as data stored on Amazon S3 buckets.

You can assign an instance profile to a layer's instances that grants permissions to apps running on those instances to access other AWS resources.

- How to manage user-based SSH keys and connect to instances.

Administrative users can assign each IAM user a personal SSH key, or authorize users to specify their own key by using the **My Settings** page. Each stack has its own settings for users' SSH access and sudo privileges on the stack's instances. For each user with SSH privileges, AWS OpsWorks creates a Linux system user and installs the user's public key on every instance that the user can access. An IAM user can then use his or her personal key to connect to instances using the built-in MindTerm SSH client or a preferred SSH client. If a user changes the personal key on his or her **My Settings** page, AWS OpsWorks automatically updates the key on every instance that the user can access.

## Tip

Another aspect of security is [Amazon EC2 security groups](#), which control network traffic to and from your instances. AWS OpsWorks provides default security groups, or you can specify custom groups. For more information, see [Create a New Stack \(p. 41\)](#).

## Topics

- [Managing User Permissions \(p. 277\)](#)
- [Signing in as an IAM User \(p. 291\)](#)
- [Setting an IAM User's Public SSH Key \(p. 292\)](#)
- [Allowing AWS OpsWorks to Act on Your Behalf \(p. 293\)](#)

- Specifying Permissions for Apps Running on EC2 instances (p. 296)

## Managing User Permissions

One way to handle AWS OpsWorks permissions is to attach to every user an IAM policy that is based on the **AWS OpsWorks Full Access** template. However, this policy allows users to perform every AWS OpsWorks action on every stack. It is often desirable instead to restrict AWS OpsWorks users to a specified set of actions or set of stack resources. You can control AWS OpsWorks user permissions in two ways: By using the AWS OpsWorks **Permissions** page and by attaching an appropriate IAM policy.

The OpsWorks **Permissions** page—or the equivalent CLI or API actions—allows you to control user permissions in a multiuser environment on a per-stack basis by assigning each user one of several *permission levels*. Each level grants permissions for a standard set of actions for a particular stack resource. Using the OpsWorks **Permissions** page, you can control the following:

- Who can access each stack.
- Which actions each user is allowed to perform on each stack.

For example, you can allow some users to only view the stack while others can deploy applications, add instances, and so on.

- Who can manage each stack.

You can delegate management of each stack to one or more specified users.

- Who has user-level SSH access and sudo privileges on each stack's Amazon EC2 instances.

You can instantly grant or remove these permissions separately for each user.

You can also use the IAM console or API to attach policies to your users that grant explicit permissions for the various AWS OpsWorks resources and actions. Using an IAM policy to specify permissions is more flexible than using the permissions levels. It also allows you to set up **IAM groups**, which grant permissions to groups of users, or define **roles that can be associated with federated users**. An IAM policy is the only way to grant permissions for certain key AWS OpsWorks actions, such as creating or cloning stacks.

The two approaches are not mutually exclusive, and it is sometimes useful to combine them; AWS OpsWorks then evaluates both sets of permissions. For example, suppose you want to allow users to add or delete instances, but not add or delete layers. None of the permission levels on the AWS OpsWorks **Permissions** page support that specific set of permissions. However, you can use the **Permissions** page to grant users a **Manage** permission level, which allows them to perform most stack operations, and then attach an IAM policy that denies permissions to add or remove layers. For more information, see [Overview of AWS IAM Permissions](#).

The following is a typical model for managing user permissions. In each case, the reader (you) is assumed to be an administrative user.

1. Use the [IAM console](#) to attach AWS OpsWorks Full Access policies to one or more administrative users.
2. Create an IAM user for each nonadministrative user with a policy that grants no AWS OpsWorks permissions.

If a user requires access only to AWS OpsWorks, you might not need to attach a policy at all. You can instead manage those permissions with the AWS OpsWorks **Permissions** page.

3. Use the AWS OpsWorks **Users** page to import the nonadministrative users into AWS OpsWorks.
4. For each stack, use the stack's **Permissions** page to assign a permission level to each user.
5. As needed, customize users' permission levels by attaching an appropriately configured IAM policy.

### Important

As a best practice, don't use root (account owner) credentials to perform everyday work in AWS. Instead, create an IAM administrators group with appropriate permissions. Then create IAM users for the people in your organization who need to perform administrative tasks (including for yourself), and add those users to the administrative group. For more information, see [IAM Best Practices](#) in the *Using IAM* guide.

### Topics

- [Managing Users \(p. 278\)](#)
- [Managing Users' Permissions Using a Stack's Permissions Page \(p. 282\)](#)
- [Managing Permissions by Attaching an IAM policy \(p. 284\)](#)
- [Example Policies \(p. 286\)](#)
- [AWS OpsWorks Permissions Levels \(p. 290\)](#)

## Managing Users

Before you can import users into AWS OpsWorks and grant them permissions, you must first have created an IAM user for each individual. To create IAM users, start by signing in to AWS as an IAM user that has been granted the permissions defined in the **IAM Full Access** policy template, or as the account owner. You then use the IAM console to [create IAM users \(p. 279\)](#) for everyone who needs to access AWS OpsWorks. You should import those users into AWS OpsWorks and then grant user permissions as follows:

### Regular AWS OpsWorks Users

Regular users don't require an attached policy. If they do have one, it typically does not include any AWS OpsWorks permissions. Instead, you use the AWS OpsWorks **Permissions** page to assign one of the following permissions levels to regular users on a stack-by-stack basis.

- **Show** permissions allow users to view the stack, but not perform any operations.
- **Deploy** permissions include the **Show** permissions and also allow users to deploy and update apps.
- **Manage** permissions include the **Deploy** permissions and also allow users to perform stack management operations such as adding layers or instances, use the **Permissions** page to set user permissions, and enable their own SSH and sudo privileges.
- **Deny** permissions deny access to the stack.

If these permissions levels are not quite what you want for a particular user, you can customize the user's permissions by attaching an IAM policy. For example, you might want to use the AWS OpsWorks **Permissions** page to assign **Manage** permissions level to a user, which grants them permissions to perform all stack management operations, but not to create or clone stacks. You could then attach a policy that restricts those permissions by denying them permission to add or delete layers or augments those permissions by allowing them to create or clone stacks. For more information, see [Managing Permissions by Attaching an IAM policy \(p. 284\)](#).

### AWS OpsWorks Administrative Users

Administrative users are the account owner or an IAM user with the same permissions that are defined by the AWS OpsWorks Full Access policy. In addition to the permissions granted to **Manage** users, this policy includes permissions for actions that cannot be granted through the **Permissions** page, such as the following:

- Importing users into AWS OpsWorks
- Creating and cloning stacks

For the complete policy, see [Example Policies \(p. 286\)](#). For a detailed list of permissions that can be granted to users only by attaching an IAM policy, see [AWS OpsWorks Permissions Levels \(p. 290\)](#).

### Topics

- [Creating an AWS OpsWorks Administrative User \(p. 279\)](#)
- [Creating IAM users for AWS OpsWorks \(p. 279\)](#)
- [Importing Users into AWS OpsWorks \(p. 280\)](#)
- [Editing User Settings \(p. 280\)](#)

## Creating an AWS OpsWorks Administrative User

You can create an AWS OpsWorks administrative user by attaching an IAM policy to a user that grants AWS OpsWorks Full Access permissions.

### To create an AWS OpsWorks administrative user

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane, click **Users** and, if necessary, click **Create New Users** to create a new IAM user for the administrative user.
3. Select the user, open the **Permissions** section, and then click **Attach User Policy**.
4. In the **Manage User Permissions** wizard, click **Select Policy Template**, select the **AWS OpsWorks Full Access** policy template and click **Apply Policy**.

#### Note

The **AWS OpsWorks Full Access** template allows users to create and manage AWS OpsWorks stacks, but they must specify an existing IAM service role for the stack. The first user to create a stack must have additional IAM permissions that are described in [Administrative Permissions \(p. 286\)](#). When this user creates the first stack, AWS OpsWorks creates an appropriate IAM service role. Thereafter, any user with CreateStack permissions can use that role to create additional stacks. For more information, see [Allowing AWS OpsWorks to Act on Your Behalf \(p. 293\)](#).

5. Edit the attached policy to fine-tune the user's permissions, as appropriate. For example, you might want an administrative user to be able to create or delete stacks, but not import new users. For more information, see [Managing Permissions by Attaching an IAM policy \(p. 284\)](#).

If you have multiple administrative users, instead of setting permissions separately for each user, you can attach a full access policy to a group and add the users to that group.

- To create a new group, click **Groups** in the navigation pane, and then click **Create New Group**. In the **Create New Group Wizard**, name your group and specify either the **AWS OpsWorks Full Access** or **Administrator Access** template.
- To add full AWS OpsWorks permissions to an existing group, click **Groups** in the navigation pane, select the group, and click the **Permissions** tab. Then click **Attach Policy** or **Attach Another Policy**. In the **Manage Group Permissions** wizard, select the **AWS OpsWorks Full Access**.

## Creating IAM users for AWS OpsWorks

Before you can import IAM users into AWS OpsWorks, you need to create them. You can do this using the [IAM console](#), command line, or API. For full instructions, see [Adding an IAM User in Your AWS Account](#).

Note that unlike [administrative users \(p. 279\)](#), you don't need to attach a policy to define permissions. You can set permissions after [importing the users into AWS OpsWorks \(p. 280\)](#), as explained in [Managing User Permissions \(p. 277\)](#).

For more information on creating IAM users and groups, see [IAM Getting Started](#).

## Importing Users into AWS OpsWorks

Administrative users can import IAM users into AWS OpsWorks.

### Note

If an IAM user has an attached policy that allows them to perform at least some AWS OpsWorks actions, they can access AWS OpsWorks without being explicitly imported. In that case, AWS OpsWorks automatically imports them. For example, a new administrative user could sign into AWS OpsWorks and open or create a stack. They perform various AWS OpsWorks actions in the process and are automatically imported.

### To import users into AWS OpsWorks

1. Sign in to AWS OpsWorks as an administrative user or as the account owner.
2. Click **Users** on the upper right to open the **Users** page.

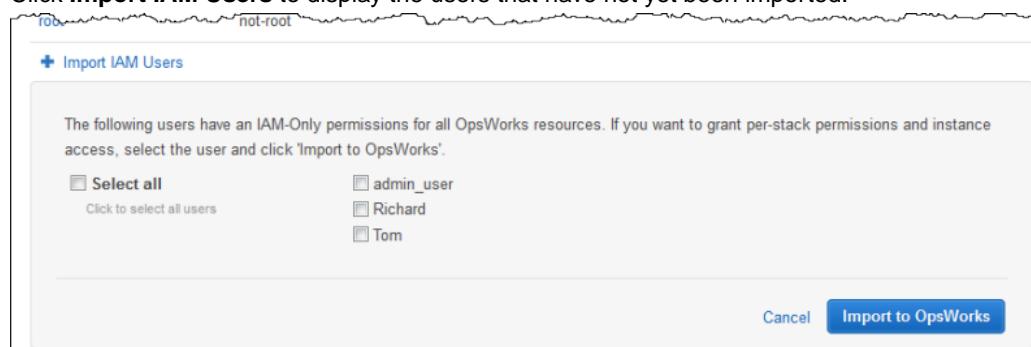
## Users

The Users page allows you to import IAM (Identity and Access Management) users. You can then use the Permissions page grant them access to specified stacks, and specify their permissions. This page can be accessed only by an AWS account owner or a user with appropriate IAM permissions. To create users, you need to go to [the IAM console](#).

Name	SSH Username	Self Management	Actions
Harold	harold	✓	<a href="#">edit</a> <a href="#">delete</a>
Maude	maude	—	<a href="#">edit</a> <a href="#">delete</a>
Charlie	charlie	—	<a href="#">edit</a>
root	not-root	—	

[+ Import IAM Users](#)

3. Click **Import IAM Users** to display the users that have not yet been imported.



4. Click **Select all** or select individual users. Then click **Import to OpsWorks**.

### Note

After you have imported an user IAM into AWS OpsWorks, if you use the IAM console or API to delete the user from your account, the user does not automatically lose SSH access you have granted through AWS OpsWorks. You must manually delete the user from AWS OpsWorks by going to the **Users** page and clicking **delete** in the user's **Actions** column.

## Editing User Settings

After you have imported users, you can edit their settings, as follows:

### To edit user settings

1. On the **Users** page, click **edit** in the user's **Actions** column.
2. You can specify the following settings.

#### Self Management

Select **Yes** to allow the user to use the MySettings page to specify his or her own public SSH key.

#### Note

You can also enable self-management by attaching an IAM policy to the user that grants permissions for the [DescribeMyUserProfile](#) and [UpdateMyUserProfile](#) actions.

#### Public SSH key

(Optional) Enter a public SSH key for the user. This key will appear on the user's **My Settings** page. For more information, see [Setting an IAM User's Public SSH Key \(p. 292\)](#). If you enable self-management, the user can specify his or her own key.

#### Permissions

(Optional) Set the user's permissions levels for each stack in one place instead of setting them separately by using each stack's **Permissions** page. For more information on permissions levels, see [Managing Users' Permissions Using a Stack's Permissions Page \(p. 282\)](#).

## User Harold

Name	Harold
ARN	arn:aws:iam::444455556666:user/Harold
Self Management	<input checked="" type="checkbox"/> Yes
SSH Username	harold
Public SSH key	<input type="text"/>

Clearing the public key will cause all SSH logins of the user to be deleted. Running processes will be terminated.

## Permissions

Stack	Permission level						Instance access	
	Deny	IAM Policies Only	Show	Deploy	Manage	SSH	sudo	
MyStack	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>	<input type="checkbox"/>	
PermStack	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="checkbox"/>	<input type="checkbox"/>	
ShortStack	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
test	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/>	<input type="checkbox"/>	

[Cancel](#) [Save](#)

# Managing Users' Permissions Using a Stack's Permissions Page

The simplest way to manage AWS OpsWorks user permissions is by using a stack's **Permissions** page. Each stack has its own page, which grants permissions for that stack.

## Permissions

[Edit](#)

Permissions specify how imported IAM (Identity and Access Management) users can access this stack. To import users, go to the [Users](#) page.  
[Learn more](#).

User Name	Permission level	SSH	sudo
Harold	Deploy	✓	-
Maude	Manage	✓	✓
Charlie	IAM Policies Only	✓	✓

### Permission levels

[Learn more](#)

- |                          |   |
|--------------------------|---|
| <b>Deny</b>              | Blocks the user's access to this stack.   |
| <b>IAM Policies Only</b> | Bases a user's permissions exclusively on policies attached to the user in IAM.   |
| <b>Show</b>              | Combines the user's IAM policies with permissions that provide read-only access to the stack's resources.                 |
| <b>Deploy</b>            | Combines the user's IAM policies with Show permissions and permissions that let the user deploy new application versions. |
| <b>Manage</b>            | Combines the user's IAM policies with permissions that provide full control of this stack.                                |

You must be signed in as an administrative user or **Manage** user to modify any of the permissions settings. The list shows only those users that have been imported into AWS OpsWorks. For information on how to create and import users, see [Managing Users \(p. 278\)](#).

The default permission level is IAM Policies Only, which grants users only those permissions that are in their attached IAM policy.

- When you import a user, he or she is added to the list for all existing stacks with an IAM Policies Only permission level.
- When you create a new stack, all current users are added to the list with **IAM Policies Only** permission levels.

### Topics

- [Setting a User's Permissions \(p. 282\)](#)
- [Viewing your Permissions \(p. 284\)](#)

## Setting a User's Permissions

### To set a user's permissions

1. In the navigation pane, click **Permissions**.
2. On the **Permissions** page, click **Edit**.
3. Change the **Permission level** and **Instance access** settings:
  - Use the **Permissions level** settings to assign one of the standard permission levels to each user, which determine whether the user can access this stack and what actions the user can perform.

If a user has an attached IAM policy, AWS OpsWorks evaluates both sets of permissions. For an example see [Example Policies \(p. 286\)](#).

- The **Instance access** settings specify whether the user has SSH and sudo privileges on the stack's instances.

The user must have an SSH public key before you can select **SSH** and you must select **SSH** before you can select **sudo**. An administrative user can either [assign a public key \(p. 280\)](#) to the user, or enable self-management to allow the user to [specify his or her own public key \(p. 292\)](#).

## Permissions

User Name	Permission level						Instance access	
	Deny	IAM Policies Only	Show	Deploy	Manage	SSH	sudo	
Harold	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Maude	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Charlie	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

Cancel Save

### Permission levels

[Learn more](#)

<b>Deny</b>	Blocks the user's access to this stack.
<b>IAM Policies Only</b>	Bases a user's permissions exclusively on policies attached to the user in IAM.
<b>Show</b>	Combines the user's IAM policies with permissions that provide read-only access to the stack's resources.
<b>Deploy</b>	Combines the user's IAM policies with Show permissions and permissions that let the user deploy new application versions.
<b>Manage</b>	Combines the user's IAM policies with permissions that provide full control of this stack.

You can assign each user to one of the following permissions levels. For a list of the actions that are allowed by each level, see [AWS OpsWorks Permissions Levels \(p. 290\)](#).

#### Deny

The user cannot perform any AWS OpsWorks actions on the stack, even if they have an attached IAM policy that grants AWS OpsWorks full access permissions. You might use this, for example, to deny some users access to stacks for unreleased products.

#### IAM Policies Only

The default level, which is assigned to all newly imported users, and to all users for newly created stacks. The user's permissions are completely determined by their attached IAM policy. If a user has no IAM policy, or their policy has no explicit AWS OpsWorks permissions, they cannot access the stack. Administrative users are typically assigned this level because their attached IAM policies already grant full access permissions.

#### Show

The user can view a stack, but not perform any operations. For example, managers might want to monitor an account's stacks, but would not need to deploy apps or modify the stack in any way.

#### Deploy

Includes the **Show** permissions and also allows the user to deploy apps. For example, an app developer might need to deploy updates to the stack's instances but not add layers or instances to the stack.

#### Manage

Includes the **Deploy** permissions and also allows the user to perform a variety of stack management operations, including:

- Adding or deleting layers and instances.

- Using the stack's **Permissions** page to assign permissions levels to users.
- Registering or deregistering resources.

For example, each stack can have a designated manager who is responsible for ensuring that the stack has an appropriate number and type of instances, handling package and operating system updates, and so on.

**Note**

The Manage level does not let users create or clone stacks. Those permissions must be granted by an attached IAM policy. For an example, see [Manage Permissions \(p. 288\)](#).

If the user also has an attached policy, AWS OpsWorks evaluates both sets of permissions. This allows you to assign a permission level to a user and then attach a policy to the user to restrict or augment the level's allowed actions. For example, you could attach a policy that allows a **Manage** user to create or clone stacks, or denies that user the ability to register or deregister resources. For some examples of such policies, see [Example Policies \(p. 286\)](#).

**Note**

If the user's policy allows additional actions, the result can appear to override the **Permissions** page settings. For example, if a user has an attached policy that allows the [CreateLayer](#) action but you use the **Permissions** page to specify **Deploy** permissions, the user is still allowed to create layers. The exception to this rule is the **Deny** option, which denies stack access even to users with AWS OpsWorks Full Access policies. For more information, see [Overview of AWS IAM Permissions](#).

## Viewing your Permissions

If [self-management \(p. 280\)](#) is enabled, a user can see a summary of his or her permissions levels for every stack by clicking **My Settings**, on the upper right. A user can also access **My Settings** if his or her attached policy grants permissions for the [DescribeMyUserProfile](#) and [UpdateMyUserProfile](#) actions.

### My Settings

[Edit](#)

Name	Harold
ARN	arn:aws:iam::444455556666:user/Harold
SSH user name	harold
Self Management	yes
Public SSH key	—

### Permissions

Stack	Permission level	SSH	sudo
MyStack	Show	—	—
PermStack	Deploy	✓	—
ShortStack	Manage	✓	✓

## Managing Permissions by Attaching an IAM policy

You can specify a user's AWS OpsWorks permissions by attaching an IAM policy. An attached policy is required for some permissions:

- Administrative user permissions, such as importing users.
- Permissions for some actions, such as creating or cloning a stack.

For a complete list of actions that require an attached policy, see [AWS OpsWorks Permissions Levels \(p. 290\)](#).

You can also use an attached policy to customize permission levels that were granted through the **Permissions** page. This section provides a brief summary of how to edit a user's IAM policy to specify AWS OpsWorks permissions. For more information, see [Permissions and Policies](#).

An IAM policy is a JSON object that contains one or more *statements*. Each statement element has a list of permissions, which have three basic elements of their own:

**Action**

The actions that the permission affects. You specify AWS OpsWorks actions as `opsworks:action`.

An Action can be set to a specific action such as `opsworks:CreateStack`, which specifies whether the user is allowed to call `CreateStack`. You can also use wildcards to specify groups of actions. For example, `opsworks:Create*` specifies all creation actions. For a complete list of AWS OpsWorks actions, see the [AWS OpsWorks API Reference](#).

**Effect**

Whether the specified actions are allowed or denied.

**Resource**

The AWS resources that the permission affects. AWS OpsWorks has one resource type, the stack. To specify permissions for a particular stack resource, set Resource to the stack's ARN, which has the following format: `arn:aws:opsworks:region:account_id:stack/stack_id/`.

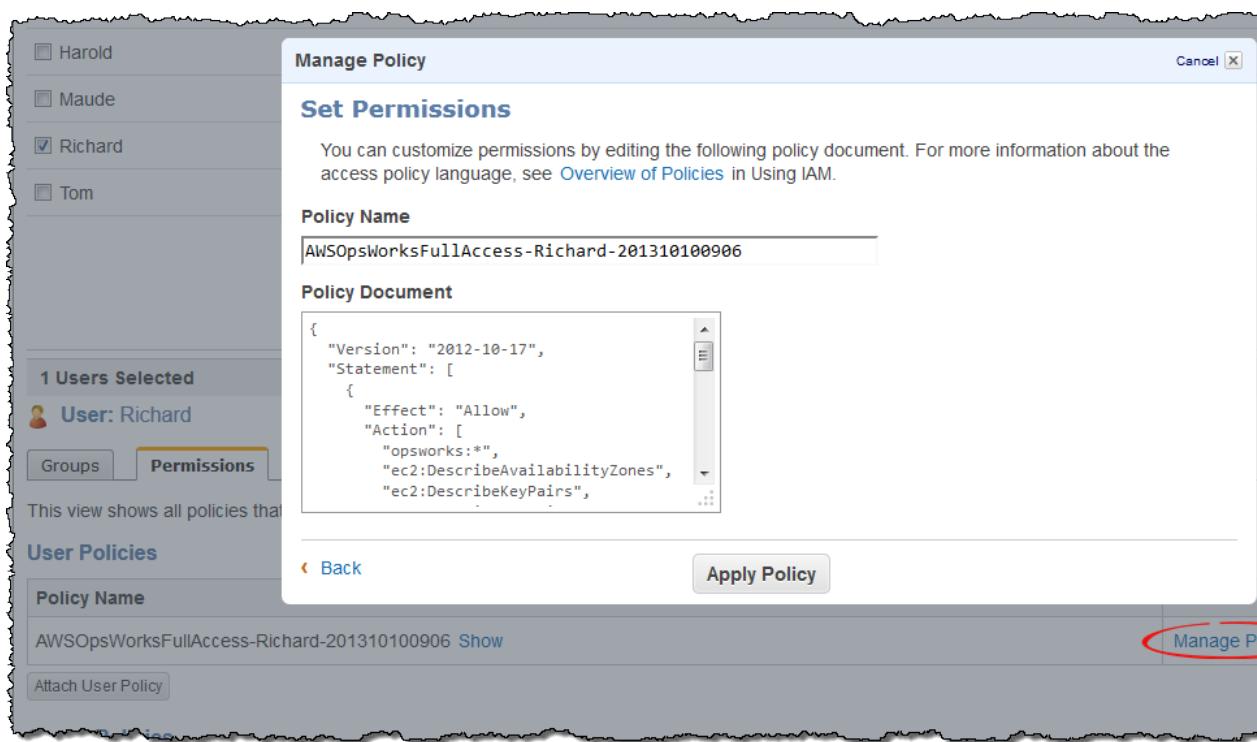
You can also use wildcards. For example, setting Resource to `*` grants permissions for every resource.

For example, the following policy denies the user the ability to stop instances on the stack whose ID is `2860-2f18b4cb-4de5-4429-a149-ff7da9f0d8ee`.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": "opsworks:StopInstance",  
            "Effect": "Deny",  
            "Resource": "arn:aws:opsworks:*.*:stack/2f18b4cb-4de5-4429-a149-  
ff7da9f0d8ee/*"  
        }  
    ]  
}
```

**To edit an IAM user's policy**

1. Go to the [IAM console](#).
2. Click **Users** in the navigation pane. Then select the user and click the Permissions tab.
3. Click **Attach User Policy** to attach a new policy or **Manage Policy** to modify an existing policy, which opens the **Set Permissions** dialog box.



4. Edit the text in the **Policy Document** box to modify the policy.

For more information on how to create or modify IAM policies, see [Permissions and Policies](#). For some examples of AWS OpsWorks policies, see [Example Policies \(p. 286\)](#).

## Example Policies

This section shows some examples of IAM policies that can be attached to AWS OpsWorks users.

- [Administrative Permissions \(p. 286\)](#) shows two policies that can be used to grant permissions to administrative users.
- [Manage Permissions \(p. 288\)](#) and [Deploy Permissions \(p. 289\)](#) show examples of policies that can be attached to a user to augment or restrict the Manage and Deploy permissions levels.

AWS OpsWorks determines the user's permissions by evaluating the permissions granted by attached IAM policies as well as the permissions granted by the Permissions page. For more information, see [Overview of AWS IAM Permissions](#). For more information on the Permissions page permissions, see [AWS OpsWorks Permissions Levels \(p. 290\)](#).

## Administrative Permissions

The following is the AWS OpsWorks Full Access policy, which can be attached to a user to grant them permissions to perform all AWS OpsWorks actions. The IAM permissions are required, among other things, to allow an administrative user to import users.

```
{
    "Version": "2012-10-17",
```

```

"Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "opsworks:*",
            "ec2:DescribeAvailabilityZones",
            "ec2:DescribeKeyPairs",
            "ec2:DescribeSecurityGroups",
            "ec2:DescribeAccountAttributes",
            "ec2:DescribeAvailabilityZones",
            "ec2:DescribeSecurityGroups",
            "ec2:DescribeSubnets",
            "ec2:DescribeVpcs",
            "elasticloadbalancing:DescribeInstanceHealth",
            "elasticloadbalancing:DescribeLoadBalancers",
            "iam:GetRolePolicy",
            "iam>ListRoles",
            "iam>ListInstanceProfiles",
            "iam:PassRole",
            "iam>ListUsers"
        ],
        "Resource": "*"
    }
]
}

```

You must create an [IAM role](#) that allows AWS OpsWorks to act on your behalf to access other AWS resources, such as Amazon EC2 instances. You typically handle this task by having an administrative user create the first stack, and letting AWS OpsWorks create the role for you. You can then use that role for all subsequent stacks. For more information, see [Allowing AWS OpsWorks to Act on Your Behalf \(p. 293\)](#).

The administrative user who creates the first stack must have permissions for some IAM actions that are not included in the AWS OpsWorks Full Access template, as follows:

```

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "opsworks:*",
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeKeyPairs",
                "ec2:DescribeAccountAttributes",
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcs",
                "elasticloadbalancing:DescribeInstanceHealth",
                "elasticloadbalancing:DescribeLoadBalancers",
                "iam:GetRolePolicy",
                "iam>ListRoles",
                "iam>ListInstanceProfiles",
                "iam:PassRole",
                "iam>ListUsers",
                "iam:PutRolePolicy",
                "iam:UpdateRole"
            ],
            "Resource": "*"
        }
    ]
}

```

```
        "iam:AddRoleToInstanceProfile",
        "iam>CreateInstanceProfile",
        "iam>CreateRole"
    ],
    "Resource": "*"
}
]
```

## Manage Permissions

The **Manage** permissions level allows a user to perform a variety of stack management actions, including adding or deleting layers. This topic describes several policies that you can attach to **Manage** users to augment or restrict the standard permissions.

Deny a **Manage** user the ability to add or delete layers

You can restrict the **Manage** permissions level to allow a user perform all **Manage** actions except adding or deleting layers by attaching the following IAM policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "opsworks:CreateLayer",
                "opsworks>DeleteLayer"
            ],
            "Resource": "*"
        }
    ]
}
```

Allow a **Manage** user to create or clone stacks

The **Manage** permissions level doesn't allow users to create or clone stacks. You can augment the **Manage** permissions to allow a user to create or clone stacks by attaching the following IAM policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "opsworks:CreateStack",
                "opsworks:CloneStack"
            ],
            "Resource": "*"
        }
    ]
}
```

Deny a **Manage** user the ability to register or deregister resources

The **Manage** permissions level allows the user to [register and deregister Amazon EBS and Elastic IP address resources](#) (p. 219) with the stack. You can restrict the **Manage** permissions to allow the user to perform all **Manage** actions except registering resources by attaching the following policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Deny",  
            "Action": [  
                "opsworks:RegisterVolume",  
                "opsworks:RegisterElasticIp"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

Allow a **Manage** user to import users

The **Manage** permissions level doesn't allow users to import users into AWS OpsWorks. You can augment the **Manage** permissions to allow a user to import and delete users by attaching the following IAM policy.

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "iam:GetRolePolicy",  
                "iam>ListRoles",  
                "iam>ListInstanceProfiles",  
                "iam>ListUsers",  
                "iam:PassRole",  
                "iam>ListInstanceProfiles",  
                "opsworks:DescribeUserProfiles",  
                "opsworks>CreateUserProfile",  
                "opsworks>DeleteUserProfile"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

## Deploy Permissions

The **Deploy** permissions level doesn't allow users to create or delete apps. You can augment the **Deploy** permissions to allow a user to create and delete apps by attaching the following IAM policy.

```
{
```

```
"Version": "2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "opsworks:CreateApp",
            "opsworks:DeleteApp"
        ],
        "Resource": "*"
    }
]
```

## AWS OpsWorks Permissions Levels

This section lists the actions that are allowed by the **Show**, **Deploy**, and **Manage** permissions levels on the AWS OpsWorks **Permissions** page. It also includes a list of actions that you can grant permissions only by attaching an IAM policy to the user.

### Show

The **Show** level allows most `DescribeXYZ` commands, with the following exceptions:

- `DescribePermissions`
- `DescribeUserProfiles`
- `DescribeMyUserProfile`

If an administrative user has enabled self-management for the user, **Show** users can also use `DescribeMyUserProfile` and `UpdateMyUserProfile`. For more information on self management, see [Editing User Settings \(p. 280\)](#).

### Deploy

The following actions are allowed by the **Deploy** level, in addition to the actions allowed by the **Show** level.

- `CreateDeployment`
- `UpdateApp`

### Manage

The following actions are allowed by the **Manage** level, in addition to the actions allowed by the **Deploy** and **Show** levels.

- `AssignVolume`
- `AssociateElasticIp`
- `AttachElasticLoadBalancer`
- `CreateApp`
- `CreateInstance`
- `CreateLayer`
- `DeleteApp`
- `DeleteInstance`
- `DeleteLayer`
- `DeleteStack`
- `DeregisterElasticIp`
- `DeregisterRdsDbInstance`
- `DeregisterVolume`
- `DescribePermissions`
- `DetachElasticLoadBalancer`

- DisassociateElasticIp
- GetHostnameSuggestion
- RebootInstance
- RegisterElasticIp
- RegisterRdsDbInstance
- RegisterVolume
- SetLoadBasedAutoScaling
- SetPermission
- SetTimeBasedAutoScaling
- StartInstance
- StartStack
- StopInstance
- StopStack
- UnassignVolume
- UpdateElasticIp
- UpdateInstance
- UpdateLayer
- UpdateRdsDbInstance
- UpdateStack
- UpdateVolume

#### Permissions That Require an IAM Policy

You must grant permissions for the following actions by attaching an appropriate IAM policy to the user. For some examples, see [Example Policies \(p. 286\)](#).

- CloneStack
- CreateStack
- CreateUserProfile
- DeleteUserProfile
- DescribeUserProfiles
- UpdateUserProfile

## Signing in as an IAM User

Each account has a URL called an account alias, which IAM users use to sign in to the account. The URL is on your IAM console's **Dashboard (Getting Started)** page under **Account Alias**.

When creating a new IAM user, an administrative user specifies or has the IAM service generate the following:

- A user name
- An optional password

You can specify a password or have IAM generate one for you.

- An optional Access Key ID and Secret Access Key

IAM generates a new key pair for each user. The keys are required for users who need to access AWS OpsWorks using the API or CLI. They are not required for console-only users.

The administrative user then provides this information and the account alias to each new user, for example, in an e-mail.

#### To sign in to AWS OpsWorks as an IAM user

1. In your browser, navigate to the account alias URL.
2. Enter your account name, user name, and password and click **Sign in**.
3. In the navigation bar, click **Services** and select **OpsWorks**.

To make AWS OpsWorks your default service, click the cube icon in the navigation bar. Then select **OpsWorks** in the **Select Start Page** dropdown.

## Setting an IAM User's Public SSH Key

Each IAM user can have a public SSH key registered with AWS OpsWorks. AWS OpsWorks then creates a Linux system user and distributes the public key to all instances that the user can access. If a user changes the public key, AWS OpsWorks automatically updates the key on every instance that the user can access. Users with public keys can be granted privileges on a per-stack basis to use the key to connect to instances by using the built-in MindTerm SSH client. For more information on how to grant privileges to use SSH, see [Setting a User's Permissions \(p. 282\)](#). For more information on how to use SSH to connect to instances, see [Using SSH to Communicate with an Instance \(p. 119\)](#).

There are two ways to register a user's public SSH key:

- An administrative user can assign a public SSH key to one or more users and provide them with the corresponding private key.
- An administrative user can enable self-management for one or more users.

Those users can then specify their own public SSH key.

The following describes how a user with self-management enabled can specify a public key. For more information on self-management and on how administrative users can assign public keys to users, see [Editing User Settings \(p. 280\)](#).

#### To specify an IAM User's SSH public key

1. Create an SSH key pair.

Note that IAM users cannot use an Amazon EC2 key pair for this purpose; Amazon EC2 does not provide the public key. Instead the user must create an SSH key pair using a third-party tool and store the public and private keys. For more information see [How to Generate Your Own Key and Import It to Amazon EC2](#).

##### Tip

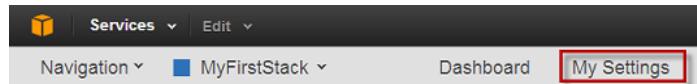
If you use [PuTTYgen](#) to generate your key pair, copy the public key from the **Public key for pasting into OpenSSH authorized\_keys file** box. Clicking **Save Public Key** saves the public key in a format that is not supported by MindTerm.

2. Sign into the AWS OpsWorks console as an IAM user with self-management enabled.

##### Important

If you sign in as an account owner, or as an IAM user that does not have self-management enabled, AWS OpsWorks does not display **My Settings**. If you are an administrative user or the account owner, you can instead specify SSH keys by going to the **Users** page and [editing the user settings \(p. 280\)](#).

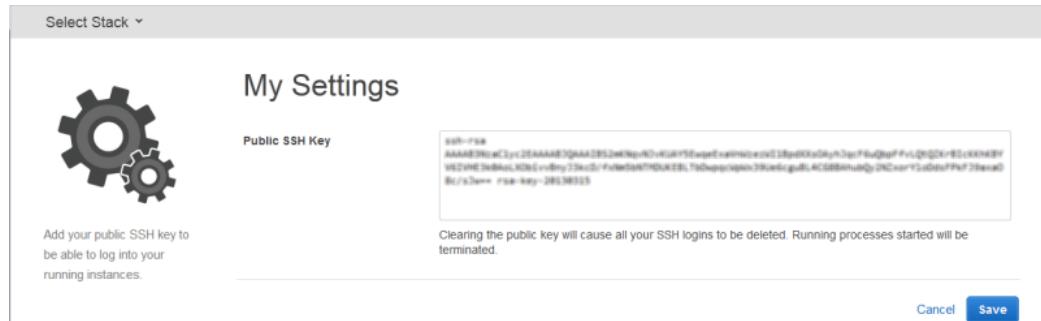
3. Select **My Settings**, which displays the settings for the signed-in IAM user.



4. On the **My Settings** page, click **Edit**.



5. In the **Public SSH Key** box, enter your SSH public key, and then click **Save**.



**Important**

To use the built-in MindTerm SSH client to connect to Amazon EC2 instances, a user must be signed in as an IAM user and have a public SSH key registered with AWS OpsWorks. For more information, see [Using the MindTerm SSH Client \(p. 120\)](#).

## Allowing AWS OpsWorks to Act on Your Behalf

AWS OpsWorks needs to interact with a variety of AWS services on your behalf. For example, AWS OpsWorks interacts with Amazon EC2 to create instances and with Amazon CloudWatch to obtain monitoring statistics. When you create a stack, you specify an IAM role, usually called a service role, that grants AWS OpsWorks the appropriate permissions.

Add Stack

Name	ShortStack
Region	US East (N. Virginia)
VPC	No VPC
Default Availability Zone	us-east-1a
Default operating system	<input checked="" type="radio"/> Amazon Linux <input type="radio"/> Ubuntu 12.04 LTS <input type="radio"/> Use custom AMI
Default root device type	<input checked="" type="radio"/> Instance store <input type="radio"/> EBS backed
IAM role	aws-opsworks-service-role

When you specify a new stack's service role, you can do one of the following:

- Have AWS OpsWorks create a new service role with a standard set of permissions.  
The role will be named something like `aws-opsworks-service-role`.
- Specify a standard service role that you created earlier.

You can usually create a standard service role when you create your first stack, and then use that role for all subsequent stacks.

- Specify a custom service role that you created by using the IAM console or API.

This approach is useful if you want to grant AWS OpsWorks more limited permissions than the standard service role.

#### Note

To create the first stack in the console, you must be signed in as an IAM user who has been granted the permissions defined in the IAM **Administrator Access** policy template or signed with the account's access and secret keys. This gives you permissions to have AWS OpsWorks create a new role. It also gives you permissions to import users, [as described earlier \(p. 280\)](#). After the first stack has been created, users can select a role created earlier by AWS OpsWorks and therefore don't require full administrative user permissions to create a stack.

The standard service role grants the following permissions:

- Perform all Amazon EC2 actions (`ec2:*`).
- Get CloudWatch statistics (`cloudwatch:GetMetricStatistics`).
- Use Elastic Load Balancing to distribute traffic to servers (`elasticloadbalancing:*`).
- Use an Amazon RDS instance as a database server (`rds:*`).
- Use IAM roles (`iam:PassRole`) to provide secure communication between AWS OpsWorks and your Amazon EC2 instances.

If you create a custom service role, you must ensure that it grants all the permissions that AWS OpsWorks needs to manage your stack.

A service role also has a trust relationship. Service roles created by AWS OpsWorks have the following trust relationship.

```
{  
    "Version": "2008-10-17",  
    "Statement": [  
        {  
            "Sid": "",  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "opsworks.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

The service role must have this trust relationship for AWS OpsWorks to act on your behalf. If you use the default service role, do not modify the trust relationship. If you are creating a custom service role, specify the trust relationship as follows:

- If you are using the **Create Role** wizard in the [IAM console](#), specify the **AWS Opsworks** role type under **AWS Service Roles** on the wizard's second page.
- If you are using a AWS CloudFormation template, you can add something like the following to your template's **Resources** section.

```
"Resources": {  
    "OpsWorksServiceRole": {  
        "Type": "AWS::IAM::Role",  
        "Properties": {  
            "AssumeRolePolicyDocument": {  
                "Statement": [ {  
                    "Effect": "Allow",  
                    "Principal": {  
                        "Service": [ "opsworks.amazonaws.com" ]  
                    },  
                    "Action": [ "sts:AssumeRole" ]  
                } ]  
            },  
            "Path": "/",  
            "Policies": [ {  
                "PolicyName": "opsworks-service",  
                "PolicyDocument": {  
                    ...  
                } ]  
            },  
        }  
    }  
}
```

# Specifying Permissions for Apps Running on EC2 instances

If the applications running on your stack's Amazon EC2 instances need to access other AWS resources, such as Amazon S3 buckets, they must have appropriate permissions. To confer those permissions, you use an instance profile. You can specify an instance profile for each instance when you [create an AWS OpsWorks stack \(p. 41\)](#).



You can also specify a profile for a layer's instances by [editing the layer configuration \(p. 59\)](#).

The instance profile specifies an IAM role. Applications running on the instance can assume that role to access AWS resources, subject to the permissions that are granted by the role's policy. For more information about how an application assumes a role, see [Assuming the Role Using an API Call](#).

You can create an instance profile in any of the following ways:

- Have AWS OpsWorks create a new profile when you create a stack.

The profile will be named something like `aws-opsworks-ec2-role` and will have a trust relationship but no policy.

- Use the IAM console or API to create a profile.

For more information, see [Roles \(Delegation and Federation\)](#).

- Use an AWS CloudFormation template to create a profile.

For some examples of how to include IAM resources in a template, see [Identity and Access Management \(IAM\) Template Snippets](#).

An instance profile must have a trust relationship and an attached policy that grants permissions to access AWS resources. Instance profiles created by AWS OpsWorks have the following trust relationship.

```
{  
    "Version": "2008-10-17",  
    "Statement": [  
        {  
            "Sid": "",  
            "Effect": "Allow",  
            "Principal": {  
                "Service": "ec2.amazonaws.com"  
            },  
            "Action": "sts:AssumeRole"  
        }  
    ]  
}
```

```
[  
]}
```

The instance profile must have this trust relationship for AWS OpsWorks to act on your behalf. If you use the default service role, do not modify the trust relationship. If you are creating a custom service role, specify the trust relationship as follows:

- If you are using the **Create Role** wizard in the [IAM console](#), specify the **Amazon EC2** role type under **AWS Service Roles** on the wizard's second page.
- If you are using a AWS CloudFormation template, you can add something like the following to your template's **Resources** section.

```
"Resources": {  
    "OpsWorksEC2Role": {  
        "Type": "AWS::IAM::Role",  
        "Properties": {  
            "AssumeRolePolicyDocument": {  
                "Statement": [ {  
                    "Effect": "Allow",  
                    "Principal": {  
                        "Service": [ "ec2.amazonaws.com" ]  
                    },  
                    "Action": [ "sts:AssumeRole" ]  
                } ]  
            },  
            "Path": "/"  
        }  
    },  
    "RootInstanceProfile": {  
        "Type": "AWS::IAM::InstanceProfile",  
        "Properties": {  
            "Path": "/",  
            "Roles": [ {  
                "Ref": "OpsWorksEC2Role"  
            } ]  
        }  
    }  
}
```

If you create your own instance profile, you can attach an appropriate policy to the profile's role at that time. If you have AWS OpsWorks create an instance profile for you when you create the stack, it does not have an attached policy and grants no permissions. After you have created the stack, you must use the [IAM console](#) or API to attach an appropriate policy to the profile's role. For example, the following policy grants full access to any S3 bucket.

```
{  
    "Version": "2012-10-17",  
    "Statement": [ {  
        "Effect": "Allow",  
        "Action": "s3:*",  
        "Resource": "*"
```

```
    }
]
}
```

For an example of how to create and use an instance profile, see [Using an Amazon S3 Bucket](#).

If your application uses an instance profile to call the AWS OpsWorks API from an EC2 instance, the policy must allow the `iam:PassRole` action in addition to the appropriate actions for AWS OpsWorks and other AWS services. The `iam:PassRole` permission allows AWS OpsWorks to assume the service role on your behalf. For more information about the AWS OpsWorks API, see [AWS OpsWorks API Reference](#).

The following is an example of an IAM policy that allows you to call any AWS OpsWorks action from an EC2 instance, as well as any Amazon EC2 or Amazon S3 action.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "ec2:*", "s3:*", "opsworks:*", "iam:PassRole" ],
      "Resource": "*"
    }
  ]
}
```

**Note**

If you do not allow `iam:PassRole`, any attempt to call an AWS OpsWorks action fails with an error like the following:

```
User: arn:aws:sts::123456789012:federated-user/Bob is not authorized
to perform: iam:PassRole on resource:
arn:aws:sts::123456789012:role/OpsWorksStackIamRole
```

For more information about using roles on an EC2 instance for permissions, see [Granting Applications that Run on Amazon EC2 Instances Access to AWS Resources](#) in the *AWS Identity and Access Management Using IAM* guide.

# Using AWS OpsWorks with Other AWS Services

---

You can have application servers running in an AWS OpsWorks stack use a variety of AWS services that are not directly integrated with AWS OpsWorks. For example, you can have your application servers use Amazon RDS as a back-end database. You can access such services by using the following general pattern:

1. Create and configure the AWS service by using the AWS console, API, or CLI and record any required configuration data that the application will need to access the service, such as host name or port.
2. Create one or more custom recipes to configure the application so that it can access the service.

The recipe obtains the configuration data from [stack configuration and deployment JSON \(p. 262\)](#) attributes that you define with custom JSON prior to running the recipes.

3. Assign the custom recipe to the Deploy lifecycle event on the application server layer.
4. Create a custom JSON object that assigns appropriate values to the configuration data attributes and add it to your stack configuration and deployment JSON.
5. Deploy the application to the stack.

Deployment runs the custom recipes, which use the configuration data values that you defined in the custom JSON to configure the application so that it can access the service.

This chapter describes how to have AWS OpsWorks application servers access a variety of AWS services. It assumes that you are already familiar with Chef cookbooks and how recipes can use stack and configuration JSON attributes to configure applications, typically by creating configuration files. If not, you should first read [Cookbooks and Recipes \(p. 153\)](#) and [Customizing AWS OpsWorks \(p. 230\)](#).

## Topics

- [Using Other Back-end Data Stores \(p. 300\)](#)
- [Using ElastiCache Redis as an In-Memory Key-Value Store \(p. 305\)](#)
- [Using an Amazon S3 Bucket \(p. 313\)](#)

# Using Other Back-end Data Stores

Application server stacks commonly include a database server to provide a back-end data store. The AWS OpsWorks MySQL layer provides integrated support for MySQL and Amazon RDS database servers. However, you can easily customize a stack to have the application servers use other database servers such as Amazon DynamoDB or MongoDB. This topic describes the basic procedure for connecting an application server to a AWS database server. As an example, it uses the stack and application from [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#) to show how to connect a PHP application server to an RDS database. For an example of how to incorporate a MongoDB database server into a stack, see [Deploying MongoDB with OpsWorks](#)

## Topics

- [How to Set up a Database Connection \(p. 300\)](#)
- [How to Connect an Application Server Instance to Amazon RDS \(p. 301\)](#)

## How to Set up a Database Connection

You set up the connection between an application server and its back-end database by using a custom recipe. The recipe configures the application server as required, typically by creating a configuration file. The recipe gets the connection data such as the host and database name from a set of attributes in the [stack configuration and deployment JSON \(p. 262\)](#) that AWS OpsWorks installs on every instance.

For example, Step 2 of [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#) is based on a stack named MyStack with two layers, PHP App Server and MySQL, each with one instance. You deploy an app named SimplePHPApp to the PHP App Server instance that uses the database on the MySQL instance as a back-end data store. When you deploy the application, AWS OpsWorks installs a stack configuration and deployment JSON that contains the database connection information, as shown in the following excerpt from a typical JSON:

```
{  
  ...  
  "deploy": {  
    "simplephpapp": {  
      ...  
      "database": {  
        "reconnect": true,  
        "password": null,  
        "username": "root",  
        "host": null,  
        "database": "simplephpapp"  
      },  
      ...  
    }  
  }  
}
```

The attribute values are supplied by AWS OpsWorks, and are either generated or based on user provided information.

To allow SimplePHPApp to access the data store, you must set up the connection between the PHP application server and the MySQL data store by assigning a custom recipe named `appsetup.rb` to the PHP App Server layer's Deploy [lifecycle event \(p. 176\)](#). When you deploy SimplePHPApp, AWS OpsWorks

runs `appsetup.rb`, which creates a configuration file named `db-connect.php` that sets up the connection, as shown in the following excerpt.

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

    variables(
      :host => (deploy[:database][:host] rescue nil),
      :user => (deploy[:database][:username] rescue nil),
      :password => (deploy[:database][:password] rescue nil),
      :db => (deploy[:database][:database] rescue nil),
      :table => (node[:phpapp][:dbtable] rescue nil)
    )
    ...
  end
end
```

The variables that characterize the connection—`host`, `user`, and so on—are set the corresponding values from the [deploy JSON's \(p. 265\)](#) `[:deploy][:app_name][:database]` attributes. For simplicity, the example assumes that you have already created a table named `urler`, so the table name is represented by `[:phpapp][:dbtable]` in the cookbook's attributes file.

This recipe can actually connect the PHP application server to any MySQL database server, not just members of a MySQL layer. To use a different MySQL server, you just have to set the `[:database]` attributes to values that are appropriate for your server, which you can do by using [custom JSON \(p. 53\)](#). AWS OpsWorks then incorporates those attributes and values into the stack configuration and deployment JSON and `appsetup.rb` uses them to create the template that sets up the connection. For more information on overriding stack configuration and deployment JSON, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

## How to Connect an Application Server Instance to Amazon RDS

This section describes how to customize MyStack from [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#) to have the PHP application server connect to an RDS instance.

### Topics

- [Create an Amazon RDS MySQL Database \(p. 302\)](#)
- [Customize the Stack to Connect to the RDS Database \(p. 303\)](#)

## Create an Amazon RDS MySQL Database

Before you create an Amazon RDS database to use with [this example \(p. 9\)](#), you must first create a database security group and configure it to allow access from the appropriate IP addresses.

### To create a database security group

1. Open the Amazon RDS console at <https://console.aws.amazon.com/rds/>.
2. Click **Security Groups** in the navigation pane and then click **Create DB Security Group**.
3. Assign a name to the security group—the example uses `phpsecgroup`—and click **Yes, Create**.
4. Select the `phpsecgroup` group in the table and click **Edit**.
5. Specify the group's CIDR/IP values. For the example, you can use a single CIDR/IP that is set to `0.0.0.0/0`, which accepts requests from any IP address. When you're done, click **Add**.

### Tip

For production use, a database security group usually limits access to only those IP addresses that need to access the database. It typically includes the addresses of the systems that you use to manage the database and the AWS OpsWorks instances that need to access the database. AWS OpsWorks automatically creates an Amazon EC2 security group for each type of layer the first time you add the layer to a stack. A simple way to provide access for AWS OpsWorks instances is to add the appropriate layer's security group to the database security group, as follows:

1. Click **Security Groups** in the navigation pane, select the appropriate group, and then click **Edit**.
2. Under **Connection Type**, select **EC2 Security Groups**.
3. Select the layer's security group from the **EC2 Security Group** list in the **Details** column and click **Add**.

For example, to provide database access to PHP App Server instances, select **AWS-OpsWorks-PHP-App-Server (`security_group_id`)**.

Now you're ready to create an RDS database for the example using the Amazon RDS console's Launch DB Instance Wizard. The following procedure is a brief summary of the essential details. For a detailed description of how to create a database, see [Getting Started with Amazon RDS](#).

### To create the Amazon RDS database

1. Click **RDS Dashboard** in the navigation pane and then click **Launch a DB Instance**.
2. Select the **MySQL Community Edition** as the DB instance.
3. Select **No, this instance...**, which is sufficient for the example. For production use, you might want to select **Yes, use Multi-AZ Deployment....** Click **Next Step**.
4. On the **DB Instance Details** page, specify the following settings:
  - **DB Instance Class:** `db.t1.micro`
  - **Multi-AZ Deployment:** **No**
  - **Allocated Storage:** 5 GB
  - **DB Instance Identifier:** `rdsexample`
  - **Master Username:** `opsworksuser`
  - **Master Password:** Specify a suitable password and record it for later use.

Accept the default settings for the other options and click **Next Step**.

5. On the **Additional Configuration** page, specify the following settings:

- **Database Name:** `rdsexampledb`
- **DB Security Group(s):** `phpsecgroup`

Accept the default settings for the other options and click **Next Step**.

6. On the **Management Options** page, set **Enabled Automatic Backups** to **No**, click **Next Step**, and then **Launch DB Instance**.
7. After the RDS instance is created, click it in the list of DB instances to see its details. Record the endpoint for later use. It will be something like `rdsexample.c6c8mntzhgv0.us-west-2.rds.amazonaws.com:3306`. Just record the DNS name; you won't need the port number.
8. Use a tool such as MySQL Workbench to create a table named `urler` in the `rdsexampledb` database by using following SQL command:

```
CREATE TABLE urler(id INT UNSIGNED NOT NULL AUTO_INCREMENT,author VARCHAR(63) NOT NULL,message TEXT,PRIMARY KEY (id))
```

## Customize the Stack to Connect to the RDS Database

Once you have [created an RDS instance \(p. 302\)](#) to use as a back-end database for the PHP application server, you can customize MyStack from [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#).

### To connect the PHP App Server to an RDS database

1. Open the AWS OpsWorks console and create a stack with a PHP App Server layer that contains one instance and deploy SimplePHPApp, as described in [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#). This stack uses version1 of SimplePHPApp, which does not use a database connection.
2. [Update the stack configuration \(p. 50\)](#) to use the custom cookbooks that include the `appsetup.rb` recipe, and related template and attribute files.
  1. Set **Use custom Chef cookbooks** to **Yes**.
  2. Set **Repository type** to **Git** and **Repository URL** to `git://github.com/amazonwebservices/opsworks-example-cookbooks.git`.
3. Add the following to the stack's **Custom Chef JSON** box to assign the RDS connection data to the `[:database]` attributes that `appsetup.rb` uses to create the configuration file.

```
{  
  "deploy": {  
    "simplephpapp": {  
      "database": {  
        "username": "opsworksuser",  
        "password": "your_password",  
        "database": "rdsexampledb",  
        "host": "rds_endpoint",  
        "adapter": "mysql"  
      }  
    }  
  }  
}
```

Use the following attribute values:

- **username**: The master user name that you specified when you created the RDS instance.

This example uses `opsworksuser`.

- **password**: The master password that you specified when you created the RDS instance.

Fill in the password that you specified.

- **database**: The database that you created when you created the RDS instance.

This example uses `rdsexampledb`.

- **host**: The RDS instance's endpoint, which you got from the RDS console when you created the instance in the previous section. Don't include the port number.

- **adapter**: The adapter.

The RDS instance for this example uses MySQL, so **adapter** is set to `mysql`. Unlike the other attributes, **adapter** is not used by `appsetup.rb`. It is instead used by the PHP App Server layer's built-in Configure recipe to create a different configuration file.

4. [Edit the SimplePHPApp configuration \(p. 134\)](#) to specify a version of SimplePHPApp that uses a back-end database, as follows:

- **Document root**: Set this option to `web`.
- **Branch/Revision**: Set this option to `version2`.

Leave the remaining options unchanged.

5. [Edit the PHP App Server layer \(p. 59\)](#) to set up the database connection by adding `phpapp::appsetup` to the layer's Deploy recipes.
6. [Deploy the new SimplePHPApp version \(p. 132\)](#).
7. When SimplePHPApp is deployed, run the application by going to the **Instances** page and clicking the `php-app1` instance's public IP address. You should see the following page in your browser, which allows you to enter text and store it in the database.



**Note**

If your stack has a MySQL layer, AWS OpsWorks automatically assigns the corresponding connection data to the [ :database] attributes. However, if you assign custom JSON to the stack that defines different [ :database] values, they override the default values. Because the stack JSON is installed on every instance for every lifecycle event, any recipes that depend on the [ :database] attributes will use the custom connection data, not the data for the MySQL layer. If you want only a particular application server layer to use the custom connection data, you can assign the custom JSON to the layer's Deploy event, and restrict that deployment to the application server layer. For more information on how to use deployment JSON, see [Deploying Apps \(p. 132\)](#). For more information on overriding AWS OpsWorks default attributes, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

## Using ElastiCache Redis as an In-Memory Key-Value Store

You can often improve application server performance by using a caching server to provide an in-memory key-value store for small items of data such as strings. Amazon ElastiCache is an AWS service that makes it easy to provide caching support for your application server, using either the [Memcached](#) or [Redis](#) caching engines. AWS OpsWorks provides built-in support for [Memcached \(p. 100\)](#). However, if Redis better suits your requirements, you can customize your stack so that your application servers use ElastiCache Redis.

This topic walks you through basic process of providing ElastiCache Redis caching support for AWS OpsWorks application servers, using a Rails application server as an example. It assumes that you already have an appropriate Ruby on Rails application. For more information on ElastiCache, see [What Is Amazon ElastiCache?](#).

**Topics**

- [Step 1: Create an ElastiCache Redis Cluster \(p. 305\)](#)
- [Step 2: Set up a Rails Stack \(p. 307\)](#)
- [Step 3: Create and Deploy a Custom Cookbook \(p. 307\)](#)
- [Step 4: Assign the Recipe to a LifeCycle Event \(p. 310\)](#)
- [Step 5: Add Access Information to the Stack Configuration JSON \(p. 311\)](#)
- [Step 6: Deploy and run the App \(p. 312\)](#)

## Step 1: Create an ElastiCache Redis Cluster

You must first create an Amazon ElastiCache Redis cluster by using the ElastiCache console, API, or CLI. The following describes how to use the console to create a cluster.

**To create an ElastiCache Redis cluster**

1. Go to the [ElastiCache console](#) and click **Launch Cache Cluster** to start the Cache Cluster wizard.
2. On the Cache Cluster Details page, do the following:
  - Set **Name** to your cache server name.  
This example uses OpsWorks-Redis.
  - Set **Engine** to **redis**.
  - Set **Topic for SNS Notification** to **Disable Notifications**.
  - Accept the defaults for the other settings and click **Continue**.

**Launch Cache Cluster Wizard**

**CACHE CLUSTER DETAILS**    ADDITIONAL CONFIGURATION    REVIEW    **Cancel**

To get started, provide the details for your Cache Cluster below.

Name: <b>*</b>	OpsWorks-Redis
Engine:	redis
Cache Engine Version:	2.6.13
Node Type:	cache.m1.small (1.3 GB me...)
Number of Nodes: <b>*</b>	1
Cache Port: <b>*</b>	6379 (e.g. 11211)
Cache Subnet Group:	Not in VPC
Preferred Zone:	No Preference
Topic for SNS Notification:	Disable Notifications <b>Manual ARN input</b>
S3 Snapshot Location:	myBucket/myFolder/o/
<b>Auto Minor Version Upgrade:</b> <input checked="" type="radio"/> Yes <input type="radio"/> No	
Note: "Auto Minor Version Upgrade" only applies to the Cache Engine software. Critical System Software patches (e.g. security related) may be applied irrespective of this selection.	
* Required	
<b>Continue</b> 	

3. On the Additional Configuration page, accept the defaults and click Continue.

**Launch Cache Cluster Wizard**

**CACHE CLUSTER DETAILS**    **ADDITIONAL CONFIGURATION**    REVIEW    **Cancel**

**Security Group**

A Cache Security Group acts like a firewall that controls network access to your Cache Clusters. Please select one or more Cache Security Groups for this Cache Cluster.

Cache Security Group(s):	default
--------------------------	---------

**Cache Parameter Group**

A Cache Parameter Group acts as a "container" for engine configuration values that can be applied to one or more Cache Clusters. If you have created a custom Cache Parameter Group you want to use, select it from below, otherwise proceed with the **default** one we created for you.

Cache Parameter Group:	default.redis2.6
------------------------	------------------

**Maintenance Window**

Maintenance Window allows you to specify the time range (UTC) during which any scheduled maintenance activities such as software patching or pending Cache Cluster modifications you requested would occur. Scheduled maintenance activities occur infrequently (generally once every few months) and will be announced on the AWS forum two weeks prior to being scheduled.

Maintenance Window:	<input checked="" type="radio"/> No Preference <input type="radio"/> Select Window
---------------------	--

**Back**    **Continue**     \* Required

4. Click Launch Cache Cluster to create the cluster.

**Important**

The default cache security group is sufficient for this example, but for production use you should create one that is appropriate for your environment. For more information, see [Managing Cache Security Groups](#).

- After the cluster has started, click the name to open the details page and click the **Nodes** tab. Record the cluster's **Port** and **Endpoint** values for later use.

Cache Cluster: opsworks-redis					
<input type="button" value="Description"/> <input checked="" type="button" value="Nodes"/> <input type="button" value="Add Nodes"/> <input type="button" value="Edit Pending Nodes"/> <input type="button" value="Reboot Nodes"/> <input type="button" value="Copy Node Endpoint(s)"/>					
◀ ◀ 1 to 1 of 1 ▶ ▶					
▲	Node Id	Node Status	Created on	Port	Endpoint
<input type="checkbox"/>	0001	available	Thu Sep 05 16:32:45 GMT-700 2013	6379	opsworks-redis.b47jtf.0001.use1.cache.amazonaws.com

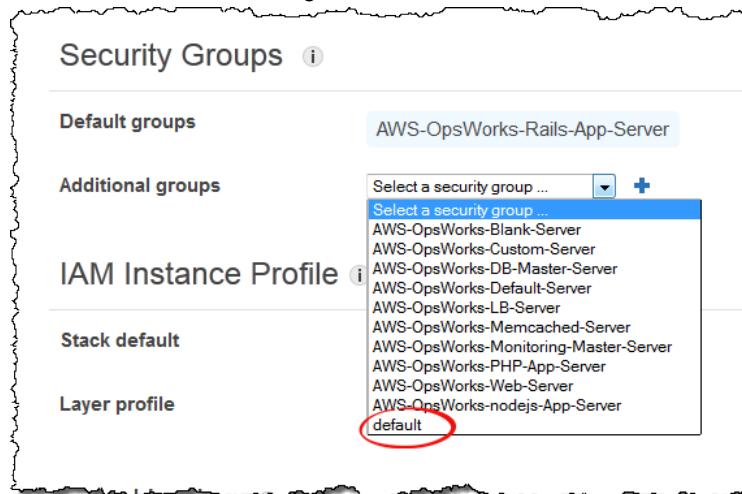
Parameter Group Status: in-sync

## Step 2: Set up a Rails Stack

In addition to creating a stack that supports a Rails App Server layer, you must also configure the layer's security groups so that the Rails server can communicate properly with Redis server.

### To set up a stack

- Create a new stack—named **RedisStack** for this example—and add a Rails App Server layer. You can use the default settings for both. For more information, see [Create a New Stack \(p. 41\)](#) and [How to Create an OpsWorks Layer \(p. 58\)](#).
- On the **Layers** page, for Rails App Server, click **Security** and then click **Edit**.
- Go to the **Security Groups** section and add the ElastiCache cluster's security group to **Additional groups**. For this example, select the **default** security group, click **+** to add it to the layer, and click **Save** to save the new configuration.



- Add an instance to the Rails App Server layer and start it. For more information on how to add and start instances, see [Adding an Instance to a Layer \(p. 103\)](#).

## Step 3: Create and Deploy a Custom Cookbook

As it stands, the stack is not quite functional yet; you need to enable your application to access the Redis server. The most flexible approach is to put a YAML file with the access information in the application's `config` subfolder. The application can then get the information from the file. Using this approach, you can change the connection information without rewriting and redeploying the application. For this example, the file should be named `redis.yml` and contain the ElastiCache cluster's host name and port, as follows:

```
host: cache-cluster-hostname
port: cache-cluster-port
```

You could manually copy this file to your servers, but a better approach is to implement a Chef *recipe* to generate the file, and have AWS OpsWorks run the recipe on every server. Chef recipes are specialized Ruby applications that AWS OpsWorks uses to perform tasks on instances such as installing packages or creating configuration files. Recipes are packaged in a *cookbook*, which can contain multiple recipes and related files such as templates for configuration files. The cookbook is placed in a repository, such as GitHub, and must have a standard directory structure. If you don't yet have a custom cookbook repository, see [Cookbook Repositories \(p. 154\)](#) for information on how to set one up.

For this example, add a cookbook named `redis-config` to your cookbook repository with the following contents:

```
my_cookbook_repository
  redis-config
    recipes
      generate.rb
    templates
      default
      redis.yml.erb
```

The `recipes` folder contains a recipe named `generate.rb`, which generates the application's configuration file from `redis.yml.erb`, as follows:

```
node[:deploy].each do |app_name, deploy_config|
  # determine root folder of new app deployment
  app_root = "#{deploy_config[:deploy_to]}/current"

  # use template 'redis.yml.erb' to generate 'config/redis.yml'
  template "#{app_root}/config/redis.yml" do
    source "redis.yml.erb"
    cookbook "redis-config"

    # set mode, group and owner of generated file
    mode "0660"
    group deploy_config[:group]
    owner deploy_config[:user]

    # define variable "@redis" to be used in the ERB template
    variables(
      :redis => deploy_config[:redis] || {}
    )

    # only generate a file if there is Redis configuration
    not_if do
      deploy_config[:redis].blank?
    end
  end
end
```

The recipe depends on data from the AWS OpsWorks [stack configuration and deployment JSON \(p. 262\)](#) object, which is installed on each instance and contains detailed information about the stack and any deployed apps. The object's `deploy` node has the following structure:

```
{
  ...
  "deploy": {
    "app1": {
      "application" : "short_name",
      ...
    }
    "app2": {
      ...
    }
    ...
  }
}
```

The `deploy` node contains a set of embedded JSON objects, one for each deployed app, that is named with the app's short name. Each app object contains a set of attributes that define the app's configuration, such as the document root and application type. For a list of the `deploy` attributes, see [Deploy Attributes \(p. 394\)](#). Recipes can use Chef attribute syntax to represent stack configuration and deployment JSON values. For example, `[:deploy][:app1][:application]` represents the `app1` application's short name.

For each app in `[:deploy]`, the recipe executes the associated code block, where `deploy_config` represents the app attribute. The recipe first sets `app_root` to the app's root directory, `[:deploy][:app_name][:deploy_to]/current`. It then uses a Chef [template resource](#) to generate a configuration file from `redis.yml.erb` and place it in the `app_root/config`.

Configuration files are typically created from templates, with many if not most of the settings defined by Chef *attributes*. With attributes you can change settings using custom JSON, as described later, instead of rewriting the template file. The `redis.yml.erb` template contains the following:

```
host: <%= @redis[:host] %>
port: <%= @redis[:port] || 6379 %>
```

The `<%... %>` elements are placeholders that represent an attribute value.

- `<%= @redis[:host] %>` represents the value of `redis[:host]`, which is the cache cluster's host name.
- `<%= @redis[:port] || 6379 %>` represents the value of the `redis[:port]` or, if that attribute is not defined, the default port value, 6379.

The `template` resource works as follows:

- `source` and `cookbook` specify the template and cookbook names, respectively.
- `mode`, `group`, and `owner` give the configuration file the same access rights as the application.
- The `variables` section sets the `@redis` variable used in the template, to the application's `[:redis]` attribute value.

The `[:redis]` attribute's values are set by using custom JSON, as described later; it is not one of the standard app attributes.

- The `not_if` directive ensures that the recipe does not generate a configuration file if one already exists.

After you author the cookbook, you must deploy it to each instance's cookbook cache. This operation does not run the recipe; it simply installs the new cookbook on the stack's instances. You typically run a recipe by assigning it to a layer's lifecycle event, as described later.

### To deploy your custom cookbook

- On the AWS OpsWorks **Stack** page, click **Stack Settings** and then **Edit**.
- In the **Configuration Management** section, set **Use custom Chef cookbooks** to **Yes**, enter the cookbook repository information, and click **Save** to update the stack configuration.

The screenshot shows a configuration dialog for "Use custom Chef cookbooks". It includes fields for "Repository type" (set to "Git"), "Repository URL" (containing "git://github.com/amazonwebservices/opsworks-cookbook"), and "Repository SSH key" (marked as "Optional"). There is also a field for "Branch/Revision" which is also marked as "Optional".

- On the **Stack** page, click **Run Command**, select the **Update Custom Cookbooks** stack command, and click **Update Custom Cookbooks** to install the new cookbook in the instances' cookbook caches.

### Run Command

The screenshot shows the "Run Command" dialog. Under "Settings", the "Command" dropdown is set to "Update Custom Cookbooks" (which is circled in red). In the "Instances" section, "Rails App Server" is selected, and "rails-app1" is checked. At the bottom right, there are "Cancel" and "Update Custom Cookbooks" buttons.

If you modify your cookbook, just run **Update Custom Cookbooks** again to install the updated version. For more information on this procedure, see [Installing Custom Cookbooks \(p. 171\)](#).

## Step 4: Assign the Recipe to a LifeCycle Event

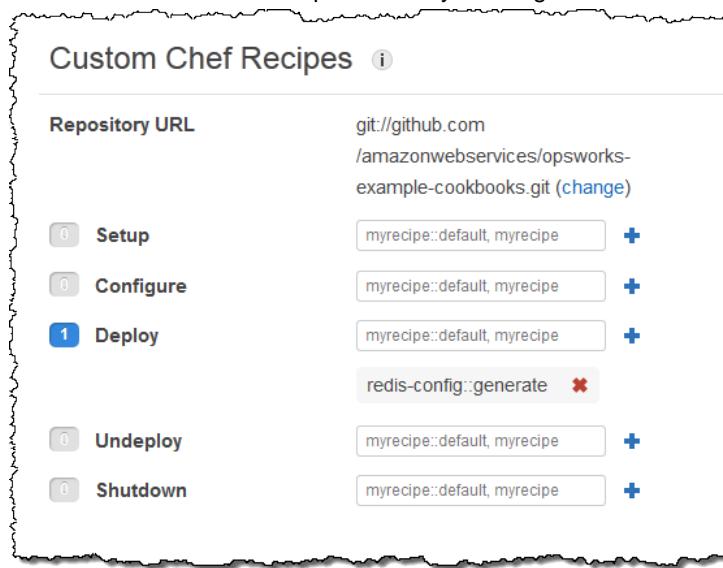
You can run custom recipes [manually \(p. 178\)](#), but the best approach is usually to have AWS OpsWorks run them automatically. Every layer has a set of built-in recipes assigned each of five [lifecycle events \(p. 176\)](#)—Setup, Configure, Deploy, Undeploy, and Shutdown. Each time an event occurs for an instance, AWS OpsWorks runs the associated recipes for each of the instance's layers, which handle the corresponding tasks. For example, when an instance finishes booting, AWS OpsWorks triggers a

Setup event. This event runs the associated layer's Setup recipes, which typically handle tasks such as installing and configuring packages.

You can have AWS OpsWorks run a custom recipe on a layer's instances by assigning the recipe to the appropriate lifecycle event. For this example, you should assign the `generate.rb` recipe to the Rails App Server layer's Deploy event. AWS OpsWorks will then run it on the layer's instances during startup, after the Setup recipes have finished, and every time you deploy an app. For more information, see [Automatically Running Recipes \(p. 177\)](#).

### To assign a recipe to the Rails App Server layer's Deploy event

1. On the AWS OpsWorks **Layers** page, for Rails App Server, click **Recipes** and then click **Edit..**.
2. Under **Custom Chef Recipes**, add the fully qualified recipe name to the deploy event and click **+**. A fully qualified recipe name uses the `cookbookname::recipename` format, where `recipename` does not include the `.rb` extension. For this example, the fully qualified name is `redis-config::generate`. Then click **Save** to update the layer configuration.



## Step 5: Add Access Information to the Stack Configuration JSON

The `generate.rb` recipe depends on a pair of stack configuration and deployment JSON attributes that represent the Redis server's host name and port. Although these attributes are part of the standard `[:deploy]` namespace, they are not automatically defined by AWS OpsWorks. Instead, you define the attributes and their values by adding a custom JSON object to the stack. The following example shows the custom JSON for this example.

### To add access information to the stack configuration and deployment JSON

1. On the AWS OpsWorks **Stack** page, click **Stack Settings** and then **Edit**.
2. In the **Configuration Management** section, add access information to the **Custom Chef JSON** box. It should look something like the following example, with these modifications:
  - Replace `elasticache_redis_example` with your app's short name.
  - Replace the `host` and `port` values with the values for the ElastiCache Redis server instance that you created in [Step 1: Create an ElastiCache Redis Cluster \(p. 305\)](#).

```
{
  "deploy": {
    "elasticache_redis_example": {
      "redis": {
        "host": "mycluster.XXXXXXXXXX.amazonaws.com",
        "port": "6379"
      }
    }
  }
}
```



The advantage of this approach is that you can change the port or host value at any time without touching your custom cookbook. AWS OpsWorks merges custom JSON into the built-in JSON and installs it on the stack's instances for all subsequent lifecycle events. Apps can then access the attribute values by using Chef node syntax, as described in [Step 3: Create and Deploy a Custom Cookbook \(p. 307\)](#). The next time you deploy an app, AWS OpsWorks will install a stack configuration and deployment JSON that contains the new definitions, and `generate.rb` will create a configuration file with the updated host and port values.

**Note**

`[:deploy]` automatically includes an attribute for every deployed app, so `[:deploy][:elasticache_redis_example]` is already in the stack and configuration JSON. However, `[:deploy][:elasticache_redis_example]` does not include a `[:redis]` attribute, defining them with custom JSON directs AWS OpsWorks to add those attributes to `[:deploy][:elasticache_redis_example]`. You can also use custom JSON to override existing attributes. For more information, see [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#).

## Step 6: Deploy and run the App

This example assumes that you have Ruby on Rails application that uses Redis. To access the configuration file, you can add the `redis` gem to your Gemfile and create a Rails initializer in `config/initializers/redis.rb` as follows:

```
REDIS_CONFIG = YAML::load_file(Rails.root.join('config', 'redis.yml'))
$redis = Redis.new(:host => REDIS_CONFIG['host'], :port => REDIS_CONFIG['port'])
```

Then [create an app \(p. 125\)](#) to represent your application and [deploy it \(p. 132\)](#) to the Rails App Server layer's instances, which updates the application code and runs `generate.rb` to generate the configuration

file. When you run the application, it will use the ElastiCache Redis instance as its in-memory key-value store.

## Using an Amazon S3 Bucket

Applications often use an Amazon S3 bucket to store large items such as images or other media files. Although AWS OpsWorks does not provide integrated support for Amazon S3, you can easily customize a stack to allow your application to use Amazon S3 storage. This topic walks you through the basic process of providing Amazon S3 access to applications, using a PHP application server as an example.

### Topics

- [Step 1: Create an Amazon S3 Bucket \(p. 313\)](#)
- [Step 2: Create a PHP App Server Stack \(p. 315\)](#)
- [Step 3: Create and Deploy a Custom Cookbook \(p. 316\)](#)
- [Step 4: Assign the Recipes to LifeCycle Events \(p. 318\)](#)
- [Step 5: Add Access Information to the Stack Configuration JSON \(p. 319\)](#)
- [Step 6: Deploy and Run PhotoApp \(p. 320\)](#)

## Step 1: Create an Amazon S3 Bucket

You must first create an Amazon S3 bucket. You can do this directly by using the Amazon S3 console, API, or CLI, but a simpler way to create resources is often to use a AWS CloudFormation template. The following template creates an Amazon S3 bucket for this example and sets up [instance profile](#) with an [IAM role](#) that grants unrestricted access to the bucket. You can then use a layer setting to attach the instance profile to the stack's application server instances, which allows the application to access the bucket, as described later. The usefulness of instance profiles isn't limited to Amazon S3; they are valuable for integrating a variety of AWS services.

```
{  
    "AWSTemplateFormatVersion" : "2010-09-09",  
    "Resources" : {  
        "AppServerRootRole": {  
            "Type": "AWS::IAM::Role",  
            "Properties": {  
                "AssumeRolePolicyDocument": {  
                    "Statement": [ {  
                        "Effect": "Allow",  
                        "Principal": {  
                            "Service": [ "ec2.amazonaws.com" ]  
                        },  
                        "Action": [ "sts:AssumeRole" ]  
                    } ]  
                },  
                "Path": "/"  
            }  
        },  
        "AppServerRolePolicies": {  
            "Type": "AWS::IAM::Policy",  
            "Properties": {  
                "PolicyName": "AppServerS3Perms",  
                "PolicyDocument": {  
                    "Version": "2012-10-17",  
                    "Statement": [ {  
                        "Effect": "Allow",  
                        "Action": [ "s3:PutObject", "s3:GetObject" ],  
                        "Resource": "arn:aws:s3:::photoapp-*"  
                    } ]  
                }  
            }  
        }  
    }  
}
```

```
        "Statement": [ {
            "Effect": "Allow",
            "Action": "s3:*",
            "Resource": { "Fn::Join" : [ "", [ "arn:aws:s3:::", { "Ref" : "AppBucket" } , /*" */ ] ]
            }
        } ]
    },
    "Roles": [ { "Ref": "AppServerRootRole" } ]
}
},
"AppServerInstanceProfile": {
    "Type": "AWS::IAM::InstanceProfile",
    "Properties": {
        "Path": "/",
        "Roles": [ { "Ref": "AppServerRootRole" } ]
    }
},
"AppBucket" : {
    "Type" : "AWS::S3::Bucket"
}
},
"Outputs" : {
    "BucketName" : {
        "Value" : { "Ref" : "AppBucket" }
    },
    "InstanceProfileName" : {
        "Value" : { "Ref" : "AppServerInstanceProfile" }
    }
}
}
```

Several things happen when you launch the template:

- The `AWS::S3::Bucket` resource creates an Amazon S3 bucket.
- The `AWS::IAM::InstanceProfile` resource creates an instance profile that will be assigned to the application server instances.
- The `AWS::IAM::Role` resource creates the instance profile's role.
- The `AWS::IAM::Policy` resource sets the role's permissions to allow unrestricted access to Amazon S3 buckets.
- The `Outputs` section displays the bucket and instance profile names in AWS CloudFormation console after you have launched the template.

You will need these values to set up your stack and app.

For more information on how to create AWS CloudFormation templates, see [Learn Template Basics](#).

### To create the Amazon S3 bucket

1. Copy the example template to a text file on your system.

This example assumes that the file is named `appserver.template`.

2. Open the [AWS CloudFormation](#) console and click **Create Stack**.
3. In the **Stack Name** box, enter the stack name.

This example assumes that the name is `AppServer`.

4. Click **Upload template file**, click **Browse**, select the `appserver.template` file that you created in Step 1, and click **Next Step**.
5. On the **Specify Parameters** page, select **I acknowledge that this template may create IAM resources**, then click **Next Step** on each page of the wizard until you reach the end. Click **Create**.
6. After the **AppServer** stack reaches **CREATE\_COMPLETE** status, select it and click its **Outputs** tab.  
  
You might need to click refresh a few times to update the status.
7. On the **Outputs** tab, record the **BucketName** and **InstanceProfileName** values for later use.

**Note**

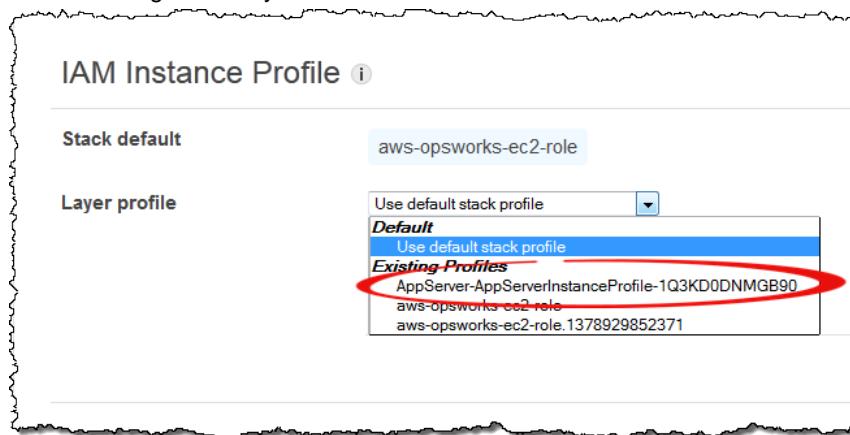
AWS CloudFormation uses the term *stack* to refer to the collection of resources that are created from a template; it is not the same as an AWS OpsWorks stack.

## Step 2: Create a PHP App Server Stack

The stack consists of two layers, PHP App Server and MySQL, each with one instance. The application stores photos on an Amazon S3 bucket, but uses the MySQL instance as a back-end data store to hold metadata for each photo.

### To create the stack

1. Create a new stack—named **PhotoSite** for this example—and add a PHP App Server layer. You can use the default settings for both. For more information, see [Create a New Stack \(p. 41\)](#) and [How to Create an OpsWorks Layer \(p. 58\)](#).
2. On the **Layers** page, for PHP App Server, click **Security** and then click **Edit**.
3. In the **Layer Profile** section, select the instance profile name that you recorded earlier, after launching the AppServer AWS CloudFormation stack. It will be something like `AppServer-AppServerInstanceProfile-1Q3KD0DNMGB90`. AWS OpsWorks assigns this profile to all of the layer's Amazon EC2 instances, which grants permission to access your Amazon S3 bucket to applications running on the layer's instances.



4. Add an instance to the PHP App Server layer and start it. For more information on how to add and start instances, see [Adding an Instance to a Layer \(p. 103\)](#).
5. Add a MySQL layer to the stack, add an instance, and start it. You can use default settings for both the layer and instance. In particular, the MySQL instance doesn't need to access the Amazon S3 bucket, so it can use the standard AWS OpsWorks instance profile, which is selected by default.

## Step 3: Create and Deploy a Custom Cookbook

The stack is not quite ready yet:

- Your application needs some information to access to the MySQL database server and the Amazon S3 bucket, such as the database host name and the Amazon S3 bucket name .
- You need to set up a database in the MySQL database server and create a table to hold the photos' metadata.

You could handle these tasks manually, but a better approach is to implement Chef *recipe* and have AWS OpsWorks run the recipe automatically on the appropriate instances. Chef recipes are specialized Ruby applications that AWS OpsWorks uses to perform tasks on instances such as installing packages or creating configuration files. They are packaged in a *cookbook*, which can contain multiple recipes and related files such as templates for configuration files. The cookbook is placed in a repository such as GitHub, and must have a standard directory structure. If you don't yet have a custom cookbook repository, see [Cookbook Repositories \(p. 154\)](#) for information on how to set one up.

For this example, the cookbook has been implemented for you and is stored in a [public GitHub repository](#). The cookbook contains two recipes, `appsetup.rb` and `dbsetup.rb`, and a template file, `db-connect.php.erb`.

The `appsetup.rb` recipe creates a configuration file that contains the information that the application needs to access the database and the Amazon S3 bucket. It is basically a lightly modified version of the `appsetup.rb` recipe described in [Connect the Application to the Database \(p. 26\)](#); see that topic for a detailed description. The primary difference is the variables that are passed to the template, which represent the access information.

```
variables(
  :host => (deploy[:database][:host] rescue nil),
  :user => (deploy[:database][:username] rescue nil),
  :password => (deploy[:database][:password] rescue nil),
  :db => (deploy[:database][:database] rescue nil),
  :table => (node[:photoapp][:dbtable] rescue nil),
  :s3bucket => (node[:photobucket] rescue nil)
)
```

The first four attributes define database connection settings, and are automatically defined by AWS OpsWorks when you create the MySQL instance.

There are two differences between these variables and the ones in the original recipe:

- Like the original recipe, the `table` variable represents the name of the database table that is created by `dbsetup.rb`, and is set to the value of an attribute that is defined in the cookbook's attributes file. However, the attribute has a different name: `[:photoapp][:dbtable]`.
- The `s3bucket` variable is specific to this example and is set to the value of an attribute that represents the Amazon S3 bucket name, `[:photobucket]`.

`[:photobucket]` is defined by using custom JSON, as described later. For more information on attributes, see [Attributes \(p. 156\)](#)

For more information on attributes, see [Attributes \(p. 156\)](#).

The `dbsetup.rb` recipe sets up a database table to hold each photo's metadata. It basically is a lightly modified version of the `dbsetup.rb` recipe described in [Set Up the Database \(p. 25\)](#); see that topic for a detailed description.

```
node[:deploy].each do |app_name, deploy|
  execute "mysql-create-table" do
    command "/usr/bin/mysql -u#{deploy[:database][:username]} -p#{deploy[:database][:password]} #{deploy[:database][:database]} -e'CREATE TABLE #{node[:photoapp][:dbtable]}(
      id INT UNSIGNED NOT NULL AUTO_INCREMENT,
      url VARCHAR(255) NOT NULL,
      caption VARCHAR(255),
      PRIMARY KEY (id)
    )'"
    not_if "/usr/bin/mysql -u#{deploy[:database][:username]} -p#{deploy[:database][:password]} #{deploy[:database][:database]} -e'SHOW TABLES' | grep #{node[:photoapp][:dbtable]}"
    action :run
  end
end
```

The only difference between this example and the original recipe is the database schema, which has three columns that contain the ID, URL, and caption of each photo that is stored on the Amazon S3 bucket.

The recipes are already implemented, so all you need to do is deploy the `photoapp` cookbook to each instance's cookbook cache. AWS OpsWorks will then run the cached recipes when the appropriate lifecycle event occurs, as described later.

### To deploy the `photoapp` cookbook

1. On the AWS OpsWorks **Stack** page, click **Stack Settings** and then **Edit**.
2. In the **Configuration Management** section:
  - Set **Use custom Chef cookbooks** to **Yes**.
  - Set **Repository type** to **Git**.
  - Set **Repository URL** to `git://github.com/amazonwebservices/opsworks-example-cookbooks.git`.
3. On the **Stack** page, click **Run Command**, select the **Update Custom Cookbooks** stack command, and click **Update Custom Cookbooks** to install the new cookbook in the instances' cookbook caches.

## Run Command

Settings

Command **Update Custom Cookbooks**

Comment  Optional Deploy comment.

[Advanced »](#)

Instances [i](#)

OpsWorks will run this command on **1 of 1** instances. The assigned recipes are run on all selected instances.

**Rails App Server**  **rails-app1** ●  
Click to select instances in this layer

[Cancel](#) [Update Custom Cookbooks](#)

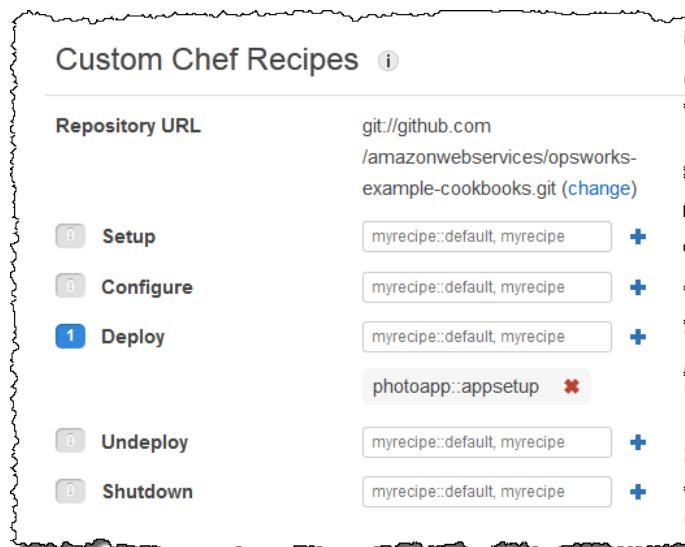
## Step 4: Assign the Recipes to LifeCycle Events

You can run custom recipes [manually \(p. 178\)](#), but the best approach is usually to have AWS OpsWorks run them automatically. Every layer has a set of built-in recipes assigned to each of five [lifecycle events \(p. 176\)](#)—Setup, Configure, Deploy, Undeploy, and Shutdown. Each time an event occurs on an instance, AWS OpsWorks runs the associated recipes for each of the instance's layers, which handle the required tasks. For example, when an instance finishes booting, AWS OpsWorks triggers a Setup event to run the Setup recipes, which typically handle tasks such as installing and configuring packages.

You can have AWS OpsWorks run custom recipes on a layer's instances by assigning each recipe to the appropriate lifecycle event. AWS OpsWorks will run any custom recipes after the layer's built-in recipes have finished. For this example, assign `appsetup.rb` to the PHP App Server layer's Deploy event and `dbsetup.rb` to the MySQL layer's Deploy event. AWS OpsWorks will then run the recipes on the associated layer's instances during startup, after the built-in Setup recipes have finished, and every time you deploy an app, after the built Deploy recipes have finished. For more information, see [Automatically Running Recipes \(p. 177\)](#).

### To assign custom recipes to the layer's Deploy event

1. On the AWS OpsWorks **Layers** page, for the PHP App Server click **Recipes** and then click **Edit**.
2. Under **Custom Chef Recipes**, add the recipe name to the deploy event and click **+**. The name must be in the Chef `cookbookname::recipename` format, where `recipename` does not include the `.rb` extension. For this example, you enter `photoapp::appsetup`. Then click **Save** to update the layer configuration.



3. On the **Layers** page, click **edit** in the MySQL layer's **Actions** column.
4. Add `photoapp::dbsetup` to the layer's Deploy event and save the new configuration.

## Step 5: Add Access Information to the Stack Configuration JSON

The `appsetup.rb` recipe depends on data from the AWS OpsWorks [stack configuration and deployment JSON \(p. 262\)](#) object, which is installed on each instance and contains detailed information about the stack and any deployed apps. The object's `deploy` node has the following structure:

```
{  
  ...  
  "deploy": {  
    "app1": {  
      "application" : "short_name",  
      ...  
    }  
    "app2": {  
      ...  
    }  
    ...  
  }  
}
```

The `deploy` node contains a set of embedded JSON objects, one for each deployed app, that is named with the app' short name. Each app object contains a set of attributes that define the app's configuration, such as the document root and app type. For a list of the deploy attributes, see [Deploy Attributes \(p. 394\)](#). You can represent stack configuration and deployment JSON values in your recipes by using Chef attribute syntax. For example, `[:deploy][:app1][:application]` represents the `app1` app's short name.

The custom recipes depend on several stack configuration and deployment JSON attributes that represent database and Amazon S3 access information:

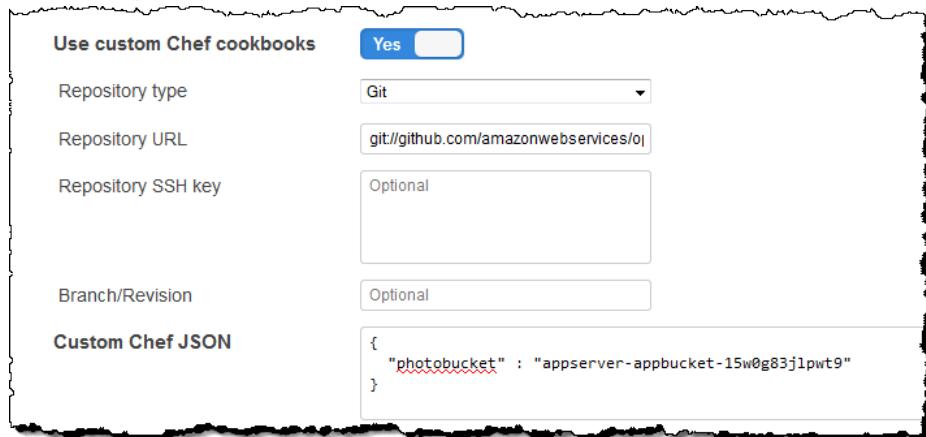
- The database connection attributes, such as [:deploy][:database][:host], are defined by AWS OpsWorks when it creates the MySQL layer.
- The table name attribute, [:photoapp][:dbtable], is defined in the custom cookbook's attributes file, and is set to foto.
- You must define the bucket name attribute, [:photobucket], by using custom JSON to add the attribute to the stack configuration and deployment JSON object.

#### To define the Amazon S3 bucket name attribute

1. On the AWS OpsWorks **Stack** page, click **Stack Settings** and then **Edit**.
2. In the **Configuration Management** section, add access information to the **Custom Chef JSON** box. It should look something like the following:

```
{  
  "photobucket" : "yourbucketname"  
}
```

Replace *yourbucketname* with the bucket name that you recorded in [Step 1: Create an Amazon S3 Bucket \(p. 313\)](#).



AWS OpsWorks merges the custom JSON into the stack configuration and deployment JSON before it installs the object on the stack's instances; `appsetup.rb` can then obtain the bucket name from the [:photobucket] attribute. If you want to change the bucket, you don't need to touch the recipe; you can just modify the stack and configuration JSON.

## Step 6: Deploy and Run PhotoApp

For this example, the application has also been implemented for you and is stored in a [public GitHub repository](#). You just need to add the app to the stack, deploy it to the application servers, and run it.

#### To add the app to the stack and deploy it to the application servers

1. Open the **Apps** page and click **Add an app**.
2. On the **Add App** page, do the following:
  - Set **Name** to **PhotoApp**.
  - Set **App type** to **PHP**.

- Set **Document root** to `web`.
- Set **Repository type** to **Git**.
- Set **Repository URL** to `git://github.com/amazonwebservices/opsworks-demo-php-photo-share-app.git`.
- Click **Add App** to accept the defaults for the other settings.

3. On the **Apps** page, click **deploy** in the PhotoApp app's **Actions** column

## Apps

An app represents code stored in a repository that you want to install on application server instances. When you deploy the app, OpsWorks downloads the code from the repository to the specified server instances. [Learn more](#).

Name	Type	Last Deployment	Actions
<a href="#">PhotoApp</a>	PHP	2013-09-27 17:38:35 UTC	<a href="#">deploy</a> <a href="#">edit</a> <a href="#">delete</a>
<a href="#">+</a> App			

4. Accept the defaults and click **Deploy** to deploy the app to the server.

To run PhotoApp, go to the **Instances** page and click the PHP App Server instance's public IP address.

PHP App Server						
Hostname	Status	Size	Type	AZ	Public IP	Actions
<a href="#">php-app1</a>	online	c1.medium	24/7	us-east-1a	54.242.100.116	<a href="#">stop</a>
<a href="#">+</a> Instance						

You should see the following user interface. Click **Add a Photo** to store a photo on the Amazon S3 bucket and the metadata in the back-end data store.



# Using the AWS OpsWorks CLI

---

The AWS OpsWorks command line interface (CLI) provides the same functionality as the console and can be used for a variety of tasks. The AWS OpsWorks CLI is part of the AWS CLI. For more information, including how to install and configure the AWS CLI, go to [What Is the AWS Command Line Interface?](#). For a complete description of each command, go to the [AWS OpsWorks reference](#).

AWS OpsWorks commands have the following general format:

```
C:\>aws --region us-east-1 opsworks command-name [--argument1 value] [...]
```

If an argument value is a JSON object, you should escape the " characters or the command might return an error that the JSON is invalid. For example, if the JSON object is `{"somekey": "somevalue"}`, you should format it as `"{\\"somekey\\":\\"somevalue\\"}"`.

Most commands return one or more values, packaged as a JSON object. The following sections contain some examples. For a detailed description of the return values for each command, go to the [AWS OpsWorks reference](#).

**Tip**

AWS CLI commands must specify a region, as shown in the examples. However, AWS OpsWorks has only one endpoint, `us-east-1`, which you must use regardless of your stack's region. To simplify your AWS OpsWorks command strings, configure the CLI to specify `us-east-1` as the default region so you can omit the `--region` argument. For more information, see [Configuring the AWS Region](#).

To use a CLI command, you must have the appropriate permissions. For more information on AWS OpsWorks permissions, see [Managing User Permissions \(p. 277\)](#). To determine the permissions required for a particular command, see the command's reference page in the [AWS OpsWorks reference](#).

The following sections describe how to use the AWS OpsWorks CLI to perform a variety of common tasks.

**Topics**

- [Create an Instance \(create-instance\) \(p. 324\)](#)
- [Deploy an App \(create-deployment\) \(p. 325\)](#)
- [Describe a Stack's Apps \(describe-apps\) \(p. 326\)](#)
- [Describe a Stack's Commands \(describe-commands\) \(p. 327\)](#)
- [Describe a Stack's Deployments \(describe-deployments\) \(p. 328\)](#)
- [Describe a Stack's Elastic IP Addresses \(describe-elastic-ips\) \(p. 329\)](#)

- [Describe a Stack's Instances \(describe-instances\) \(p. 329\)](#)
- [Describe Stacks \(describe-stacks\) \(p. 330\)](#)
- [Describe a Stack's Layers \(describe-layers\) \(p. 331\)](#)
- [Execute a Recipe \(create-deployment\) \(p. 334\)](#)
- [Install Dependencies \(create-deployment\) \(p. 335\)](#)
- [Update the Stack Configuration \(update-stack\) \(p. 335\)](#)

## Create an Instance (create-instance)

Use the [create-instance](#) command to create an instance on a specified stack.

```
C:\>aws opsworks --region us-east-1 create-instance --stack-id 935450cc-61e0-4b03-a3e0-160ac817d2bb
--layer-ids 5c8c272a-f2d5-42e3-8245-5bf3927cb65b --instance-type m1.large
--os "Amazon Linux"
```

The arguments are as follows:

- **stack-id** – You can get the stack ID from the stack's settings page on the console (look for **OpsWorks ID**) or by calling [describe-stacks](#).
- **layer-ids** – You can get layer IDs from the layer's details page on the console (look for **OpsWorks ID**) or by calling [describe-layers](#). In this example, the instance belongs to only one layer. You can get a layer ID from the layer's details page on the console (look for **OpsWorks ID**) or by calling [describe-layers](#).
- **instance-type** – The specification that defines the memory, CPU, storage capacity, and hourly cost for the instance; `m1.large` for this example.
- **os** - The instance's operating system; Amazon Linux for this example.

The command returns a JSON object that contains the instance ID, as follows:

```
{
    "InstanceId": "5f9adeaa-c94c-42c6-aefc-28a5376002cd"
}
```

This example creates an instance with a default host name, which is simply an integer. The following section describes how to create an instance with a host name generated from a theme.

## Create an Instance with a Themed Host Name

You can also create an instance with a themed host name. You specify the theme when you create the stack. For more information, see [Create a New Stack \(p. 41\)](#). To create the instance, first call [get-hostname-suggestion](#) to generate a name. The following example generates a themed name for a in a PHP App Server instance.

```
C:\>aws opsworks get-hostname-suggestion --region us-east-1 --layer-id 5c8c272a-f2d5-42e3-8245-5bf3927cb65b
```

If you specify the default Layer Dependent theme, `get-hostname-suggestion` simply appends a digit to the layer's short name. For a PHP App Server instance, the short name is `php-app3` and the generated host name will be something like `php-app3`, depending on how many instances the layer already has. For more information, see [Create a New Stack \(p. 41\)](#).

The command returns the generated host name.

```
{  
    "Hostname": "php-app2",  
    "LayerId": "5c8c272a-f2d5-42e3-8245-5bf3927cb65b"  
}
```

You can then use the `hostname` argument to pass the generated name to `create-instance`, as follows:

```
c:\>aws --region us-east-1 opsworks create-instance --stack-id 935450cc-61e0-  
4b03-a3e0-160ac817d2bb  
    --layer-ids 5c8c272a-f2d5-42e3-8245-5bf3927cb65b --instance-type m1.large -  
    -os "Amazon Linux" --hostname "php-app2"
```

## Deploy an App (create-deployment)

Use the [create-deployment](#) command to deploy an app to a specified stack.

```
aws opsworks --region us-east-1 create-deployment --stack-id cfb7e082-ad1d-4599-  
8e81-de1c39ab45bf  
    --app-id 307be5c8-d55d-47b5-bd6e-7bd417c6c7eb --command "{\"Name\": \"de  
ploy\"}"
```

The arguments are as follows:

- `stack-id` – You can get the stack ID from the stack's settings page on the console (look for **OpsWorks ID**) or by calling `describe-stacks`.
- `app-id` – You can get app ID from the app's details page (look for **OpsWorks ID**) or by calling `describe-apps`.
- `command` – The argument takes a JSON object that sets the command name to `deploy`, which deploys the specified app to the stack.

Notice that the `"` characters in the JSON object are all escaped. Otherwise, the command might return an error that the JSON is invalid.

The preceding example deploys to every instance in the stack. To deploy to a specified subset of instances, add an `instance-ids` argument and list the instance IDs.

```
{  
    "DeploymentId": "5746c781-df7f-4c87-84a7-65a119880560"  
}
```

The command returns a JSON object that contains the deployment ID, as follows:

## Migrate the Rails Database

Rails apps have an optional `migrate` argument, which specifies whether to migrate the database when you deploy the app. The default setting is `false`. To migrate the database when you deploy your app, set `migrate` to `true`, as shown in the following example.

```
aws opsworks --region us-east-1 create-deployment --stack-id cfb7e082-ad1d-4599-8e81-de1c39ab45bf --app-id 307be5c8-d55d-47b5-bd6e-7bd417c6c7eb --command "{\"Name\":\"deploy\", \"Args\":{\"migrate\":[\"true\"]}}"
```

## Describe a Stack's Apps (describe-apps)

Use the [describe-apps](#) command to get details about a stack's apps.

```
aws opsworks --region us-east-1 describe-apps --stack-id 38ee91e2-abdc-4208-a107-0b7168b3cc7a
```

The preceding example returns a JSON object that contains information about each app. This example has only one app. For a description of each parameter, see [describe-apps](#).

```
{
  "Apps": [
    {
      "StackId": "38ee91e2-abdc-4208-a107-0b7168b3cc7a",
      "AppSource": {
        "Url": "https://s3-us-west-2.amazonaws.com/opsworks-tomcat/simplejsp.zip",
        "Type": "archive"
      },
      "Name": "SimpleJSP",
      "EnableSsl": false,
      "SslConfiguration": {},
      "AppId": "daldecc1-0dff-43ea-ad7c-bb667cd87c8b",
      "Attributes": {
        "RailsEnv": null,
        "AutoBundleOnDeploy": "true",
        "DocumentRoot": "ROOT"
      },
      "Shortname": "simplejsp",
      "Type": "other",
      "CreatedAt": "2013-08-01T21:46:54+00:00"
    }
  ]
}
```

# Describe a Stack's Commands (describe-commands)

Use the [describe-commands](#) command to get details about AWS OpsWorks commands. The following example gets information about the commands that have been executed on a specified instance.

```
aws opsworks --region us-east-1 describe-commands --instance-id 8c2673b9-3fe5-420d-9cfa-78d875ee7687
```

The command returns a JSON object that contains details about each command. The `Type` parameter identifies the command name, deploy or undeploy for this example. For a description of the other parameters, see [describe-commands](#).

```
{  
  "Commands": [  
    {  
      "Status": "successful",  
      "CompletedAt": "2013-07-25T18:57:47+00:00",  
      "InstanceId": "8c2673b9-3fe5-420d-9cfa-78d875ee7687",  
      "DeploymentId": "6ed0df4c-9ef7-4812-8dac-d54a05be1029",  
      "AcknowledgedAt": "2013-07-25T18:57:41+00:00",  
      "LogUrl": "https://s3.amazonaws.com/prod_stage-log/logs/008c1a91-ec59-4d51-971d-3adff54b00cc?AWSAccessKeyId=AIDACKCEVSQ6C2EXAMPLE &Expires=1375394373&Signature=HkXil6UuNfxTCC37EPQAa462E1E%3D&response-cache-control=private&response-content-encoding=gzip&response-content-type=text%2Fplain",  
  
      "Type": "undeploy",  
      "CommandId": "008c1a91-ec59-4d51-971d-3adff54b00cc",  
      "CreatedAt": "2013-07-25T18:57:34+00:00",  
      "ExitCode": 0  
    },  
    {  
      "Status": "successful",  
      "CompletedAt": "2013-07-25T18:55:40+00:00",  
      "InstanceId": "8c2673b9-3fe5-420d-9cfa-78d875ee7687",  
      "DeploymentId": "19d3121e-d949-4ff2-9f9d-94eac087862a",  
      "AcknowledgedAt": "2013-07-25T18:55:32+00:00",  
      "LogUrl": "https://s3.amazonaws.com/prod_stage-log/logs/899d3d64-0384-47b6-a586-33433aad117c?AWSAccessKeyId=AIDACKCEVSQ6C2EXAMPLE &Expires=1375394373&Signature=xMsJvtLuUqWmsr8s%2FAjVru0BtRs%3D&response-cache-control=private&response-content-encoding=gzip&response-content-type=text%2Fplain",  
      "Type": "deploy",  
      "CommandId": "899d3d64-0384-47b6-a586-33433aad117c",  
      "CreatedAt": "2013-07-25T18:55:29+00:00",  
      "ExitCode": 0  
    }  
  ]  
}
```

# Describe a Stack's Deployments (describe-deployments)

Use the [describe-deployments](#) command to get details about deployments.

```
aws opsworks --region us-east-1 describe-deployments --stack-id 38ee91e2-abdc-4208-a107-0b7168b3cc7a
```

The preceding command returns a JSON object that contains details about each deployment for the specified stack. For a description of each parameter, see [describe-deployments](#).

```
{
  "Deployments": [
    {
      "StackId": "38ee91e2-abdc-4208-a107-0b7168b3cc7a",
      "Status": "successful",
      "CompletedAt": "2013-07-25T18:57:49+00:00",
      "DeploymentId": "6ed0df4c-9ef7-4812-8dac-d54a05be1029",
      "Command": {
        "Args": {},
        "Name": "undeploy"
      },
      "CreatedAt": "2013-07-25T18:57:34+00:00",
      "Duration": 15,
      "InstanceIds": [
        "8c2673b9-3fe5-420d-9cfa-78d875ee7687",
        "9e588a25-35b2-4804-bd43-488f85ebe5b7"
      ]
    },
    {
      "StackId": "38ee91e2-abdc-4208-a107-0b7168b3cc7a",
      "Status": "successful",
      "CompletedAt": "2013-07-25T18:56:41+00:00",
      "IamUserArn": "arn:aws:iam::444455556666:user/example-user",
      "DeploymentId": "19d3121e-d949-4ff2-9f9d-94eac087862a",
      "Command": {
        "Args": {},
        "Name": "deploy"
      },
      "InstanceIds": [
        "8c2673b9-3fe5-420d-9cfa-78d875ee7687",
        "9e588a25-35b2-4804-bd43-488f85ebe5b7"
      ],
      "Duration": 72,
      "CreatedAt": "2013-07-25T18:55:29+00:00"
    }
  ]
}
```

## Describe a Stack's Elastic IP Addresses (describe-elastic-ips)

Use the [describe-elastic-ips](#) command to get details about the Elastic IP addresses that have been registered with a stack.

```
aws opsworks --region us-east-1 describe-elastic-ips --instance-id b62f3e04-e9eb-436c-a91f-d9e9a396b7b0
```

The preceding command returns a JSON object that contains details about each Elastic IP address (one in this example) for a specified instance. For a description of each parameter, see [describe-elastic-ips](#).

```
{  
    "ElasticIps": [  
        {  
            "Ip": "192.0.2.0",  
            "Domain": "standard",  
            "Region": "us-west-2"  
        }  
    ]  
}
```

## Describe a Stack's Instances (describe-instances)

Use the [describe-instances](#) command to get details about a stack's instances.

```
C:\>aws opsworks --region us-east-1 describe-instances --stack-id 38ee91e2-abdc-4208-a107-0b7168b3cc7a
```

The preceding command returns a JSON object that contains details about each instance in a specified stack. For a description of each parameter, see [describe-instances](#).

```
{  
    "Instances": [  
        {  
            "StackId": "38ee91e2-abdc-4208-a107-0b7168b3cc7a",  
            "SshHostRsaKeyFingerprint":  
"f4:3b:8e:27:1b:73:98:80:5d:d7:33:e2:b8:c8:8f:de",  
            "Status": "stopped",  
            "AvailabilityZone": "us-west-2a",  
            "SshHostDsaKeyFingerprint":  
"e8:9b:c7:02:18:2a:bd:ab:45:89:21:4e:af:0b:07:ac",  
            "InstanceId": "8c2673b9-3fe5-420d-9cfa-78d875ee7687",  
            "Os": "Amazon Linux",  
            "Hostname": "db-master1",  
            "SecurityGroupIds": [],  
            "Architecture": "x86_64",  
            "RootDeviceType": "instance-store",  
            "VolumeIds": []  
        }  
    ]  
}
```

```

        "LayerIds": [
            "41a20847-d594-4325-8447-171821916b73"
        ],
        "InstanceType": "c1.medium",
        "CreatedAt": "2013-07-25T18:11:27+00:00"
    },
    {
        "StackId": "38ee91e2-abdc-4208-a107-0b7168b3cc7a",
        "SshHostRsaKeyFingerprint":
        "ae:3a:85:54:66:f3:ce:98:d9:83:39:1e:10:a9:38:12",
        "Status": "stopped",
        "AvailabilityZone": "us-west-2a",
        "SshHostDsaKeyFingerprint":
        "5b:b9:6f:5b:1c:ec:55:85:f3:45:f1:28:25:1f:de:e4",
        "InstanceId": "9e588a25-35b2-4804-bd43-488f85ebe5b7",
        "Os": "Amazon Linux",
        "Hostname": "tomcustom1",
        "SecurityGroupIds": [],
        "Architecture": "x86_64",
        "RootDeviceType": "instance-store",
        "LayerIds": [
            "e6cbcd29-d223-40fc-8243-2eb213377440"
        ],
        "InstanceType": "c1.medium",
        "CreatedAt": "2013-07-25T18:15:52+00:00"
    }
]
}

```

## Describe Stacks (describe-stacks)

Use the [describe-stacks](#) command to get details about specified stacks.

```
aws opsworks --region us-east-1 describe-stacks
```

The preceding command returns a JSON object that contains details about each stack in the account, two in this example. For a description of each parameter, see [describe-stacks](#).

```
{
    "Stacks": [
        {
            "ServiceRoleArn": "arn:aws:iam::444455556666:role/aws-opsworks-
service-role",
            "StackId": "aeb7523e-7c8b-49d4-b866-03aae9d4fbcb",
            "DefaultRootDeviceType": "instance-store",
            "Name": "TomStack-sd",
            "ConfigurationManager": {
                "Version": "11.4",
                "Name": "Chef"
            },
            "UseCustomCookbooks": true,
            "CustomJson": "{\n    \"tomcat\": {\n        \"base_version\": 7,\n        \"java_opts\": \"-Djava.awt.headless=true -Xmx256m\",\n        \"datasources\": {\n            \"ROOT\": \"jdbc/mydb\"\n        }\n    }\n}"
        }
    ]
}
```

```
        "Region": "us-east-1",
        "DefaultInstanceProfileArn": "arn:aws:iam::444455556666:instance-
profile/aws-opsworks-ec2-role",
        "CustomCookbooksSource": {
            "Url": "git://github.com/example-repo/tomcustom.git",
            "Type": "git"
        },
        "DefaultAvailabilityZone": "us-east-1a",
        "HostnameTheme": "Layer_Dependent",
        "Attributes": {
            "Color": "rgb(45, 114, 184)"
        },
        "DefaultOs": "Amazon Linux",
        "CreatedAt": "2013-08-01T22:53:42+00:00"
    },
    {
        "ServiceRoleArn": "arn:aws:iam::444455556666:role/aws-opsworks-
service-role",
        "StackId": "40738975-da59-4c5b-9789-3e422f2cf099",
        "DefaultRootDeviceType": "instance-store",
        "Name": "MyStack",
        "ConfigurationManager": {
            "Version": "11.4",
            "Name": "Chef"
        },
        "UseCustomCookbooks": false,
        "Region": "us-east-1",
        "DefaultInstanceProfileArn": "arn:aws:iam::444455556666:instance-
profile/aws-opsworks-ec2-role",
        "CustomCookbooksSource": {},
        "DefaultAvailabilityZone": "us-east-1a",
        "HostnameTheme": "Layer_Dependent",
        "Attributes": {
            "Color": "rgb(45, 114, 184)"
        },
        "DefaultOs": "Amazon Linux",
        "CreatedAt": "2013-10-25T19:24:30+00:00"
    }
]
}
```

## Describe a Stack's Layers (describe-layers)

Use the [describe-layers](#) command to get details about specified layers.

```
aws opsworks --region us-east-1 describe-layers --stack-id 38ee91e2-abdc-4208-
a107-0b7168b3cc7a
```

The preceding command returns a JSON object that contains details about each layer in a specified stack—in this example, a MySQL layer and a custom layer. For a description of each parameter, see [describe-layers](#).

```
{  
    "Layers": [  
        {  
            "StackId": "38ee91e2-abdc-4208-a107-0b7168b3cc7a",  
            "Type": "db-master",  
            "DefaultSecurityGroupNames": [  
                "AWS-OpsWorks-DB-Master-Server"  
            ],  
            "Name": "MySQL",  
            "Packages": [],  
            "DefaultRecipes": {  
                "Undeploy": [],  
                "Setup": [  
                    "opsworks_initial_setup",  
                    "ssh_host_keys",  
                    "ssh_users",  
                    "mysql::client",  
                    "dependencies",  
                    "ebs",  
                    "opsworks_ganglia::client",  
                    "mysql::server",  
                    "dependencies",  
                    "deploy::mysql"  
                ],  
                "Configure": [  
                    "opsworks_ganglia::configure-client",  
                    "ssh_users",  
                    "agent_version",  
                    "deploy::mysql"  
                ],  
                "Shutdown": [  
                    "opsworks_shutdown::default",  
                    "mysql::stop"  
                ],  
                "Deploy": [  
                    "deploy::default",  
                    "deploy::mysql"  
                ]  
            },  
            "CustomRecipes": {  
                "Undeploy": [],  
                "Setup": [],  
                "Configure": [],  
                "Shutdown": [],  
                "Deploy": []  
            },  
            "EnableAutoHealing": false,  
            "LayerId": "41a20847-d594-4325-8447-171821916b73",  
            "Attributes": {  
                "MysqlRootPasswordUbiquitous": "true",  
                "RubygemsVersion": null,  
                "RailsStack": null,  
                "HaproxyHealthCheckMethod": null,  
                "RubyVersion": null,  
                "BundlerVersion": null,  
                "HaproxyStatsPassword": null,  
                "PassengerVersion": null,  
                "MemcachedMemory": null,  
                "MemcachedPort": null  
            }  
        }  
    ]  
}
```

```
        "EnableHaproxyStats": null,
        "ManageBundler": null,
        "NodejsVersion": null,
        "HaproxyHealthCheckUrl": null,
        "MysqlRootPassword": "*****FILTERED*****",
        "GangliaPassword": null,
        "GangliaUser": null,
        "HaproxyStatsUrl": null,
        "GangliaUrl": null,
        "HaproxyStatsUser": null
    },
    "Shortname": "db-master",
    "AutoAssignElasticIps": false,
    "CustomSecurityGroupIds": [],
    "CreatedAt": "2013-07-25T18:11:19+00:00",
    "VolumeConfigurations": [
        {
            "MountPoint": "/vol/mysql",
            "Size": 10,
            "NumberOfDisks": 1
        }
    ]
},
{
    "StackId": "38ee91e2-abdc-4208-a107-0b7168b3cc7a",
    "Type": "custom",
    "DefaultSecurityGroupNames": [
        "AWS-OpsWorks-Custom-Server"
    ],
    "Name": "TomCustom",
    "Packages": [],
    "DefaultRecipes": {
        "Undeploy": [],
        "Setup": [
            "opsworks_initial_setup",
            "ssh_host_keys",
            "ssh_users",
            "mysql::client",
            "dependencies",
            "ebs",
            "opsworks_ganglia::client"
        ],
        "Configure": [
            "opsworks_ganglia::configure-client",
            "ssh_users",
            "agent_version"
        ],
        "Shutdown": [
            "opsworks_shutdown::default"
        ],
        "Deploy": [
            "deploy::default"
        ]
    },
    "CustomRecipes": {
        "Undeploy": [],
        "Setup": [
            "tomcat::setup"
        ]
    }
}
```

```
        ],
        "Configure": [
            "tomcat::configure"
        ],
        "Shutdown": [],
        "Deploy": [
            "tomcat::deploy"
        ]
    },
    "EnableAutoHealing": true,
    "LayerId": "e6cbcd29-d223-40fc-8243-2eb213377440",
    "Attributes": {
        "MysqlRootPasswordUbiquitous": null,
        "RubygemsVersion": null,
        "RailsStack": null,
        "HaproxyHealthCheckMethod": null,
        "RubyVersion": null,
        "BundlerVersion": null,
        "HaproxyStatsPassword": null,
        "PassengerVersion": null,
        "MemcachedMemory": null,
        "EnableHaproxyStats": null,
        "ManageBundler": null,
        "NodejsVersion": null,
        "HaproxyHealthCheckUrl": null,
        "MysqlRootPassword": null,
        "GangliaPassword": null,
        "GangliaUser": null,
        "HaproxyStatsUrl": null,
        "GangliaUrl": null,
        "HaproxyStatsUser": null
    },
    "Shortname": "tomcustom",
    "AutoAssignElasticIps": false,
    "CustomSecurityGroupIds": [],
    "CreatedAt": "2013-07-25T18:12:53+00:00",
    "VolumeConfigurations": []
}
]
```

## Execute a Recipe (create-deployment)

Use the [create-deployment](#) command to execute [stack commands \(p. 52\)](#) and [deployment commands \(p. 132\)](#). The following example executes a stack command to run a custom recipe on a specified stack.

```
aws opsworks --region us-east-1 create-deployment --stack-id 935450cc-61e0-4b03-a3e0-160ac817d2bb
--command "{\"Name\":\"execute_recipes\", \"Args\":{\"recipes\":[\"phpapp::appsetup\"]}}
```

The `command` argument takes a JSON object that is formatted as follows:

- **Name** - Specifies the command name. The `execute_recipes` command used for this example executes a specified recipe on the stack's instances.
- **Args** - Specifies a list of arguments and their values. This example has one argument, `recipes`, which is set to the recipe to be executed, `phpapp::appsetup`.

Notice that the " characters in the JSON object are all escaped. Otherwise, the command might return an error that the JSON is invalid.

The command returns a deployment ID, which you can use to identify the command for other CLI commands such as `describe-commands`.

```
{  
    "DeploymentId": "5cbaar7b9-4e09-4e53-aalb-314fbd106038"  
}
```

## Install Dependencies (create-deployment)

Use the [create-deployment](#) command to execute [stack commands \(p. 52\)](#) and [deployment commands \(p. 132\)](#). The following example runs the `update_dependencies` stack command to update the dependencies on a stack's instances.

```
aws opsworks --region us-east-1 create-deployment --stack-id 935450cc-61e0-4b03-a3e0-160ac817d2bb  
--command "{\"Name\":\"install_dependencies\"}"
```

The `command` argument takes a JSON object with a `Name` parameter whose value specifies the command name, `install_dependencies` for this example. Notice that the " characters in the JSON object are all escaped. Otherwise, the command might return an error that the JSON is invalid.

The command returns a deployment ID, which you can use to identify the command for other CLI commands such as `describe-commands`.

```
{  
    "DeploymentId": "aef5b255-8604-4928-81b3-9b0187f962ff"  
}
```

## Update the Stack Configuration (update-stack)

Use the [update-stack](#) command to update the configuration of a specified stack. The following example updates a stack to add custom JSON to the [stack configuration JSON \(p. 53\)](#).

```
aws opsworks --region us-east-1 update-stack --stack-id 935450cc-61e0-4b03-a3e0-160ac817d2bb  
--custom-json "{\"somekey\":\"somevalue\"}" --service-role-arn  
arn:aws:iam::444455556666:role/aws-opsworks-service-role
```

Notice that the " characters in the JSON object are all escaped. Otherwise, the command might return an error that the JSON is invalid.

**Note**

The example also specifies a service role for the stack. You must set `service-role-arn` to a valid service role ARN or the action will fail; there is no default value. You can specify the stack's current service role ARN, if you prefer, but you must do so explicitly.

The `update-stack` command does not return a value.

# Debugging and Troubleshooting Guide

---

If you need to debug a recipe or troubleshoot a service issue, the best approach generally is to work through the following steps, in order:

1. Check [Common Debugging and Troubleshooting Issues \(p. 347\)](#) for your specific issue.
2. Search the [AWS OpsWorks Forum](#) to see if the issue has been discussed there.

The Forum includes many experienced users and is monitored by the AWS OpsWorks team.

3. For issues with custom recipes, see [Debugging Recipes \(p. 337\)](#).
4. Contact AWS OpsWorks support or post your issue on the [AWS OpsWorks Forum](#).

The following section provides guidance for debugging recipes. The final section describes common debugging and troubleshooting issues and their solutions.

## Topics

- [Debugging Recipes \(p. 337\)](#)
- [Common Debugging and Troubleshooting Issues \(p. 347\)](#)

## Debugging Recipes

When a lifecycle event occurs, or you run the [Execute Recipes stack command \(p. 52\)](#), AWS OpsWorks issues a command to the [agent \(p. 344\)](#) to initiate a [Chef Solo run](#) on the specified instances to execute the appropriate recipes, including your custom recipes. This section describes some ways that you can debug failed recipes.

## Topics

- [Chef Logs \(p. 338\)](#)
- [Using the Agent CLI \(p. 344\)](#)
- [Running Cookbook Tests \(p. 347\)](#)

## Chef Logs

Chef logs are one of your key troubleshooting resources, especially for debugging recipes. AWS OpsWorks captures the Chef log for each command, and retains the logs for an instance's thirty most recent commands. Because the run is in debug mode, the log contains a detailed description of the Chef run, including the text that is sent to `stdout` and `stderr`. If a recipe fails, the log includes the Chef stack trace.

AWS OpsWorks gives you several ways to view Chef logs. Once you have the log information, you can use it to debug failed recipes.

### Note

You can also view a specified log's tail by using SSH to connect to the instance, and running the agent CLI `show_log` command. For more information, see [Displaying Chef Logs \(p. 345\)](#).

### Topics

- [Viewing a Chef Log with the Console \(p. 338\)](#)
- [Viewing a Chef Log with the CLI or API \(p. 338\)](#)
- [Viewing a Chef Log on an Instance \(p. 339\)](#)
- [Interpreting a Chef Log \(p. 340\)](#)
- [Common Chef Log Errors \(p. 343\)](#)

## Viewing a Chef Log with the Console

The simplest way to view a Chef log is to go to the instance's details page. The **Logs** section includes an entry for each event and [Execute Recipes \(p. 52\)](#) command. The following shows an instance's **Logs** section, with **configure** and **setup** commands, which correspond to Configure and Setup lifecycle events.

Created at	Command	Duration	Log
✓ 2013-10-02 21:06:56 UTC	configure	00:01:04	<a href="#">show</a>
✓ 2013-10-02 21:01:15 UTC	setup	00:05:40	<a href="#">show</a>

Click **show** in the appropriate command's **Log** column to view the corresponding Chef log. If an error occurs, AWS OpsWorks automatically opens the log to the error, which is usually at the end of the file.

## Viewing a Chef Log with the CLI or API

You can use the AWS OpsWorks CLI `describe-commands` command, or the `DescribeCommands` API action to view the logs, which are stored on an Amazon S3 bucket. The following shows how to use the CLI to view any of the current set of logs file for a specified instance. The procedure for using `DescribeCommands` is essentially similar.

### To use the AWS OpsWorks to view an instance's Chef logs

1. Open the instance's details page and copy its **OpsWorks ID** value.
2. Use the ID value to run the `describe-commands` CLI command, as follows:

```
aws opsworks describe-commands --instance-id 67bf0da2-29ed-4217-990c-d895d51812b9
```

The command returns a JSON object with an embedded object for each command that AWS OpsWorks has executed on the instance with the most recent first. The `Type` parameter contains

the command type for each embedded object, a `configure` command and a `setup` command in this example.

```
{
    "Commands": [
        {
            "Status": "successful",
            "CompletedAt": "2013-10-25T19:38:36+00:00",
            "InstanceId": "67bf0da2-29ed-4217-990c-d895d51812b9",
            "AcknowledgedAt": "2013-10-25T19:38:24+00:00",
            "LogUrl": "https://s3.amazonaws.com/prod_stage-log/logs/b6c402df-5c23-45b2-a707-ad20b9c5ae40?AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1382731518&Signature=YkqS5IZN2P4wixjHwoC3aCMbn5s%3D&response-cache-control=private&response-content-encoding=gzip&response-content-type=text%2Fplain",
            "Type": "configure",
            "CommandId": "b6c402df-5c23-45b2-a707-ad20b9c5ae40",
            "CreatedAt": "2013-10-25T19:38:11+00:00",
            "ExitCode": 0
        },
        {
            "Status": "successful",
            "CompletedAt": "2013-10-25T19:31:08+00:00",
            "InstanceId": "67bf0da2-29ed-4217-990c-d895d51812b9",
            "AcknowledgedAt": "2013-10-25T19:29:01+00:00",
            "LogUrl": "https://s3.amazonaws.com/prod_stage-log/logs/2a90e862-f974-42a6-9342-9a4f03468358?AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1382731518&Signature=cxKYHO8mCCd4MvOyFb6ywebeQtA%3D&response-cache-control=private&response-content-encoding=gzip&response-content-type=text%2Fplain",
            "Type": "setup",
            "CommandId": "2a90e862-f974-42a6-9342-9a4f03468358",
            "CreatedAt": "2013-10-25T19:26:01+00:00",
            "ExitCode": 0
        }
    ]
}
```

3. Copy the `LogUrl` value to your browser to view the log.

If the instance has more than a few commands, you can add parameters to `describe-commands` to filter which commands are included in the response object. For more information, see [describe-commands](#).

## Viewing a Chef Log on an Instance

AWS OpsWorks stores each instance's Chef logs in its `/var/lib/aws/opsworks/chef` directory along with the associated [stack configuration and deployment JSON \(p. 262\)](#). You need [sudo privileges \(p. 277\)](#) to access this directory. The logs for each run are named with the run's time stamp. The following example shows a log and its associated JSON file.

```
-rw-r----- 1 aws aws      6158 Oct 11 19:06 2013-10-11-19-06-24-01.json
-rw-r--r-- 1 root root   129009 Oct 11 19:08 2013-10-11-19-06-24-01.log
```

**Tip**

The JSON file consists of a single long string and is not very readable. A much easier way to view a Chef run's JSON is to use the agent CLI `get_json` command, which displays a formatted version of the specified JSON. For more information, see [Displaying the Stack Configuration and Deployment JSON \(p. 346\)](#).

Opsworks stores its internal logs in the instance's `/var/log/aws/opsworks` folder. The information is usually not very helpful for troubleshooting purposes. However, these logs are useful to AWS OpsWorks support, and you might be asked to provide them if you encounter an issue with the service. The Linux logs can also sometimes provide useful troubleshooting data.

## Interpreting a Chef Log

The beginning of the log contains mostly internal Chef logging.

```
# Logfile created on Thu Oct 17 17:25:12 +0000 2013 by logger.rb/1.2.6
[2013-10-17T17:25:12+00:00] INFO: *** Chef 11.4.4 ***
[2013-10-17T17:25:13+00:00] DEBUG: Building node object for php-app1.localdomain
[2013-10-17T17:25:13+00:00] DEBUG: Extracting run list from JSON attributes
provided on command line
[2013-10-17T17:25:13+00:00] INFO: Setting the run_list to ["opsworks_custom_cook
books::load", "opsworks_custom_cookbooks::execute"] from JSON
[2013-10-17T17:25:13+00:00] DEBUG: Applying attributes from json file
[2013-10-17T17:25:13+00:00] DEBUG: Platform is amazon version 2013.03
[2013-10-17T17:25:13+00:00] INFO: Run List is [recipe[opsworks_custom_cook
books::load], recipe[opsworks_custom_cookbooks::execute]]
[2013-10-17T17:25:13+00:00] INFO: Run List expands to [opsworks_custom_cook
books::load, opsworks_custom_cookbooks::execute]
[2013-10-17T17:25:13+00:00] INFO: Starting Chef Run for php-app1.localdomain
[2013-10-17T17:25:13+00:00] INFO: Running start handlers
[2013-10-17T17:25:13+00:00] INFO: Start handlers complete.
[2013-10-17T17:25:13+00:00] DEBUG: No chefignore file found at
/opt/aws/opsworks/releases/20131015111601_209/cookbooks/chefignore no files
will be ignored
[2013-10-17T17:25:13+00:00] DEBUG: Cookbooks to compile: ["gem_support",
"packages", "opsworks_bundler", "opsworks_rubygems", "ruby", "ruby_enterprise",
"dependencies", "opsworks_commons", "scm_helper", :opsworks_custom_cookbooks]
[2013-10-17T17:25:13+00:00] DEBUG: Loading cookbook gem_support's library file:
/opt/aws/opsworks/releases/20131015111601_209/cookbooks/gem_support/librar
ies/current_gem_version.rb
[2013-10-17T17:25:13+00:00] DEBUG: Loading cookbook packages's library file:
/opt/aws/opsworks/releases/20131015111601_209/cookbooks/packages/libraries/pack
ages.rb
[2013-10-17T17:25:13+00:00] DEBUG: Loading cookbook dependencies's library file:
/opt/aws/opsworks/releases/20131015111601_209/cookbooks/dependencies/librar
ies/current_gem_version.rb
[2013-10-17T17:25:13+00:00] DEBUG: Loading cookbook opsworks_commons's library
file: /opt/aws/opsworks/releases/20131015111601_209/cookbooks/opsworks_com
mons/libraries/activesupport_blank.rb
[2013-10-17T17:25:13+00:00] DEBUG: Loading cookbook opsworks_commons's library
file: /opt/aws/opsworks/releases/20131015111601_209/cookbooks/opsworks_com
mons/libraries/monkey_patch_chefgem_resource.rb
...
```

This part of the file is useful largely to Chef experts. Note that the run list includes only two recipes, even though most commands involve many more. These two recipes handle the task of loading and executing all the other built-in and custom recipes.

The most interesting part of the file is typically at the end. If a run ends successfully, you should see something like the following:

```
...
[Tue, 11 Jun 2013 16:00:50 +0000] DEBUG: STDERR:
[Tue, 11 Jun 2013 16:00:50 +0000] DEBUG: ---- End output of /sbin/service mysqld
restart ----
[Tue, 11 Jun 2013 16:00:50 +0000] DEBUG: Ran /sbin/service mysqld restart re
turned 0
[Tue, 11 Jun 2013 16:00:50 +0000] INFO: service[mysql]: restarted successfully
[Tue, 11 Jun 2013 16:00:50 +0000] INFO: Chef Run complete in 84.07096 seconds
[Tue, 11 Jun 2013 16:00:50 +0000] INFO: cleaning the checksum cache
[Tue, 11 Jun 2013 16:00:50 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--tmp-chef-rendered-template20130611-4899-
8wef7e-0
[Tue, 11 Jun 2013 16:00:50 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--tmp-chef-rendered-template20130611-4899-
1xpwyb6-0
[Tue, 11 Jun 2013 16:00:50 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--etc-monit-conf
[Tue, 11 Jun 2013 16:00:50 +0000] INFO: Running report handlers
[Tue, 11 Jun 2013 16:00:50 +0000] INFO: Report handlers complete
[Tue, 11 Jun 2013 16:00:50 +0000] DEBUG: Exiting
```

**Tip**

You can use the agent CLI to display the log's tail during or after the run. For more information, see [Displaying Chef Logs \(p. 345\)](#).

If a recipe fails, you should look for an ERROR-level output, which will contain an exception followed by a Chef stack trace, such as the following:

```
...
Please report any problems with the /usr/scripts/mysqlbug script!

[ OK ]
MySQL Daemon failed to start.
Starting mysqld: [FAILED]STDERR: 130611 15:07:55 [Warning] The syntax '--log-
slow-queries' is deprecated and will be removed in a future release. Please use
'--slow-query-log'/'--slow-query-log-file' instead.
130611 15:07:56 [Warning] The syntax '--log-slow-queries' is deprecated and
will be removed in a future release. Please use '--slow-query-log'/'--slow-
query-log-file' instead.
---- End output of /sbin/service mysqld start ----

/opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-
0.9.15.5/bin/.../lib/chef/mixin/command.rb:184:in `handle_command_failures'
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-
0.9.15.5/bin/.../lib/chef/mixin/command.rb:131:in `run_command'
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-
0.9.15.5/bin/.../lib/chef/provider/service/init.rb:37:in `start_service'
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-
```

```
0.9.15.5/bin/../lib/chef/provider/service.rb:60:in `action_start'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/resource.rb:406:in `send'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/resource.rb:406:in `run_action'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/runner.rb:53:in `run_action'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/runner.rb:89:in `converge'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/runner.rb:89:in `each'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/runner.rb:89:in `converge'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/resource_collection.rb:94:in `execute_each_resource'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/resource_collection/stepable_iterator.rb:116:in `call'  
  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/resource_collection/stepable_iterator.rb:116:in  
`call_iterator_block'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/resource_collection/stepable_iterator.rb:85:in `step'  
  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/resource_collection/stepable_iterator.rb:104:in `iter-  
ate'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/resource_collection/stepable_iterator.rb:55:in  
`each_with_index'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/resource_collection.rb:92:in `execute_each_resource'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/runner.rb:84:in `converge'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/client.rb:268:in `converge'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/client.rb:158:in `run'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/application/solo.rb:190:in `run_application'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/application/solo.rb:181:in `loop'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/application/solo.rb:181:in `run_application'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/../lib/chef/application.rb:62:in `run'  
  /opt/aws/opsworks/releases/20130605160141_122/vendor/bundle/ruby/1.8/gems/chef-  
0.9.15.5/bin/chef-solo:25  
  /opt/aws/opsworks/current/bin/chef-solo:16:in `load'  
  /opt/aws/opsworks/current/bin/chef-solo:16
```

The end of the file is the Chef stack trace. You should also examine the output just before the exception, which often contains a system error such as package not available that can also be useful in determining the failure cause. In this case, the MySQL daemon failed to start.

## Common Chef Log Errors

The following are some common Chef log errors, and how to address them.

### Log ends abruptly

If a Chef log ends abruptly without either indicating success or displaying error information, you have probably encountered a low-memory state that prevented Chef from completing the log. Your best option is to try again with a larger instance.

### Missing cookbook or recipe

If the Chef run encounters a cookbook or recipe that is not in the cookbook cache, you will see something like the following:

```
DEBUG: Loading Recipe mycookbook::myrecipe via include_recipe
ERROR: Caught exception during execution of custom recipe: mycookbook::myrecipe:
        Cannot find a cookbook named mycookbook; did you forget to add metadata
        to a cookbook?
```

This entry indicates that the `mycookbook` cookbook is not in the cookbook cache. With Chef 11.4, you can also encounter this error if you do not declare dependencies correctly in `metadata.rb`.

AWS OpsWorks runs recipes from the instance's cookbook cache. It downloads cookbooks from your repository to this cache when the instance starts. However, AWS OpsWorks does not automatically update the cache on an online instance if you subsequently modify the cookbooks in your repository. If you have modified your cookbooks or added new cookbooks since starting the instance, take the following steps:

1. Make sure that you have committed your changes to the repository.
2. Run the [Update Cookbooks stack command \(p. 52\)](#) to update the cookbook cache with the most recent version from the repository.

### Local command failure

If a Chef `execute` resource fails to execute the specified command, you will see something like the following:

```
DEBUG: ---- End output of ./configure --with-config-file-path=/ returned 2
ERROR: execute[PHP: ./configure] (/root/opsworks-agent/site-cookbooks/php-fpm/recipes/install.rb line 48) had an error:
        ./configure --with-config-file-path=/

```

Scroll up in the log and you should see the command's `stderr` and `stdout` output, which should help you determine why the command failed.

### Package failure

If a package installation fails, you will see something like the following:

```
ERROR: package[zend-server-ce-php-5.3] (/root/opsworks-agent/site-cookbooks/zend_server/recipes/install.rb line 20)
        had an error: apt-get -q -y --force-yes install zend-server-ce-php-5.3=5.0.4+b17 returned 100, expected 0

```

Scroll up in the log and you should see the command's `STDOUT` and `STDERR` output, which should help you determine why the package installation failed.

## Using the Agent CLI

On every online instance, AWS OpsWorks installs an agent, which communicates with the service. The AWS OpsWorks service in turn sends commands to the agent to perform tasks such as initiating Chef runs on the instance when a lifecycle event occurs. The agent exposes a command line interface (CLI) that is very useful for troubleshooting. To run agent CLI commands, use SSH to connect to an instance. You can then run agent CLI commands to perform a variety of tasks, including the following:

- Execute recipes.
- Display Chef logs.
- Display [stack configuration and deployment JSON \(p. 262\)](#).

For more information on how to set up an SSH connection to an instance, see [Using SSH to Communicate with an Instance \(p. 119\)](#). You must also have [SSH and sudo permissions \(p. 277\)](#) for the stack.

This section describes how to use the agent CLI for troubleshooting. For more information and a complete command reference, see [Appendix B: Instance Agent CLI \(p. 368\)](#).

### Topics

- [Executing Recipes \(p. 344\)](#)
- [Displaying Chef Logs \(p. 345\)](#)
- [Displaying the Stack Configuration and Deployment JSON \(p. 346\)](#)

## Executing Recipes

The agent CLI [run\\_command \(p. 373\)](#) command directs the agent to rerun a command that it performed earlier. The most useful commands for troubleshooting—`setup`, `configure`, `deploy`, and `undeploy`—each correspond to a lifecycle event. They direct the agent to initiate a Chef run to execute the associated recipes.

#### Note

The `run_command` command is limited to executing the group of recipes that is associated with a specified command, typically the recipes that are associated with a lifecycle event. You cannot use it to execute a particular recipe. To execute one or more specified recipes, use the [Execute Recipes stack command \(p. 52\)](#) or the equivalent CLI or API actions (`create-deployment` and `CreateDeployment`).

The `run_command` command is quite useful for debugging custom recipes, especially recipes that are assigned to the Setup and Configure lifecycle events, which you can't trigger directly from the console. By using `run_command`, you can run a particular event's recipes as often as you need without having to start or stop instances.

#### Note

AWS OpsWorks runs recipes from the instance's cookbook cache, not the cookbook repository. AWS OpsWorks downloads cookbooks to this cache when the instance starts, but does not automatically update the cache on online instances if you subsequently modify your cookbooks. If you have modified your cookbooks since starting the instance, be sure to run the [Update Cookbooks stack command \(p. 52\)](#) stack command to update the cookbook cache with the most recent version from the repository.

The agent caches only the most recent commands. You can list them by running [list\\_commands \(p. 373\)](#), which returns a list of cached commands and the time that they were performed.

```
sudo opsworks-agent-cli list_commands
```

```
2013-02-26T19:08:26      setup
2013-02-26T19:12:01      configure
2013-02-26T19:12:05      configure
2013-02-26T19:22:12      deploy
```

To rerun the most recent command, run this:

```
sudo opsworks-agent-cli run_command
```

To rerun the most recent instance of a specified command, run this:

```
sudo opsworks-agent-cli run_command command
```

For example, to rerun the Setup recipes, you can run the following command:

```
sudo opsworks-agent-cli run_command setup
```

Each command has an associated [stack configuration and deployment JSON \(p. 262\)](#) that represents stack and deployment state at the time the command was run. Because that data can change from one command to the next, an older instance of a command might use somewhat different data than the most recent one. To rerun a particular instance of a command, copy the time from the `list_commands` output and run the following:

```
sudo opsworks-agent-cli run_command time
```

The preceding examples all rerun the command using the default JSON, which is the JSON was installed for that command. You can rerun a command against an arbitrary JSON file as follows:

```
sudo opsworks-agent-cli run_command /path/to/valid/json.file
```

## Displaying Chef Logs

The agent CLI [show\\_log \(p. 374\)](#) command displays a specified log. After the command is finished, you will be looking at the end of the file. The `show_log` command therefore provides a convenient way to tail the log, which is typically where you find error information. You can scroll up to see the earlier parts of the log.

To display the current command's log, run this:

```
sudo opsworks-agent-cli show_log
```

You can also display logs for a particular command, but be aware that the agent caches logs for only the last thirty commands. You can list an instance's commands by running [list\\_commands \(p. 373\)](#), which returns a list of cached commands and the time that they were performed. For an example, see [Executing Recipes \(p. 344\)](#).

To show the log for the most recent execution of a particular command, run the following:

```
sudo opsworks-agent-cli show_log command
```

The command parameter can be set to `setup`, `configure`, `deploy`, `undeploy`, `start`, `stop`, or `restart`. Most of these commands correspond to lifecycle events and direct the agent to run the associated recipes.

To display the log for a particular command execution, copy the date from the `list_commands` output and run:

```
sudo opsworks-agent-cli show_log date
```

If a command is still executing, `show_log` displays the log's current state.

**Tip**

One way to use `show_log` to troubleshoot errors and out-of-memory issues is to tail a log during execution, as follows:

1. Use `run_command` to trigger the appropriate lifecycle event. For more information, see [Executing Recipes \(p. 344\)](#).
2. Repeatedly run `show_log` to see the tail of the log as it is being written.

If Chef runs out of memory or exits unexpectedly, the log will end abruptly. If a recipe fails, the log will end with an exception and a stack trace.

## Displaying the Stack Configuration and Deployment JSON

Much of the data used by recipes comes from the [stack configuration and deployment JSON \(p. 262\)](#), which defines a set of Chef attributes that provide a detailed description of the stack configuration, any deployments, and optional custom attributes that users can add. For each command, AWS OpsWorks installs a JSON that represents the stack and deployment state at the time of the command . For more information, see [Stack Configuration and Deployment JSON \(p. 262\)](#).

If your custom recipes obtain data from the stack configuration and deployment JSON, you can verify the data by examining the JSON. The easiest way to display the stack configuration and deployment JSON is to run the agent CLI [get\\_json \(p. 369\)](#) command, which displays a formatted version of the JSON object. The following shows the first few lines of some typical output:

```
{  
  "opsworks": {  
    "layers": {  
      "php-app": {  
        "id": "4a2a56c8-f909-4b39-81f8-556536d20648",  
        "instances": {  
          "php-app2": {  
            "elastic_ip": null,  
            "region": "us-west-2",  
            "booted_at": "2013-02-26T20:41:10+00:00",  
            "ip": "10.112.235.192",  
            "aws_instance_id": "i-34037f06",  
            "availability_zone": "us-west-2a",  
            "instance_type": "c1.medium",  
            "private_dns_name": "ip-10-252-0-203.us-west-2.compute.internal",  
            "private_ip": "10.252.0.203",  
            "created_at": "2013-02-26T20:39:39+00:00",  
            "status": "online",  
            "backends": 8,  
            "public_dns_name": "ec2-10-112-235-192.us-west-2.compute.amazonaws.com"  
          }  
        }  
      }  
    }  
  }  
}
```

```
aws.com"  
...
```

You can display the most recent stack configuration and deployment JSON as follows:

```
sudo opsworks-agent-cli get_json
```

You can display the most recent stack configuration and deployment JSON for a specified command by executing the following:

```
sudo opsworks-agent-cli get_json command
```

The command parameter can be set to `setup`, `configure`, `deploy`, `undeploy`, `start`, `stop`, or `restart`. Most of these commands correspond to lifecycle events and direct the agent to run the associated recipes.

You can display the stack configuration and deployment JSON for a particular command execution by specifying the command's date like this:

```
sudo opsworks-agent-cli get_json date
```

The simplest way to use this command is as follows:

1. Run `list_commands`, which returns a list of commands that have been run on the instance, and the date that each command was run.
2. Copy the date for the appropriate command and use it as the `get_json date` argument.

## Running Cookbook Tests

You can run `minitest-chef-handler` tests on your Chef 11.4 cookbooks by adding the following to your custom stack JSON.

```
{"opsworks": {  
    "run_cookbook_tests": true  
}}
```

For more information on how to use custom JSON, see [Use Custom JSON to Modify the Stack Configuration JSON \(p. 53\)](#).

## Common Debugging and Troubleshooting Issues

This section describes some commonly encountered debugging and troubleshooting issues and their solutions.

### Topics

- [Debugging Custom Recipes \(p. 348\)](#)

- Troubleshooting AWS OpsWorks (p. 349)

## Debugging Custom Recipes

This section contains some commonly encountered recipe issues and their solutions.

### Topics

- [Chef Returns ResourceNotFound \(p. 348\)](#)
- [Custom Recipe has Stopped Working \(p. 348\)](#)
- [Opscode Community Cookbook Does Not Work with AWS OpsWorks \(p. 348\)](#)
- [Overridden Built-in Recipes Don't Run \(p. 348\)](#)

## Chef Returns ResourceNotFound

**Problem:** Chef returns ResourceNotFound for one of your custom recipes.

**Cause 1:** You copied a built-in recipe into a custom cookbook. Instead of redefining the recipe, Chef 11 creates a second resource with the same name. This creates a resource collection instead of a single resource, causing Chef 11 to return a ResourceNotFound message.

**Solution:** Do not copy built-in recipes into your custom cookbooks.

**Cause 2** The recipe is not in your custom cookbook, or you added the recipe to the cookbook since the last time you updated the instance's cookbook cache.

**Solution:** Make sure that the recipe has been committed to your repository and then run the [Update Cookbooks stack command \(p. 52\)](#) to update the cookbook caches with the most recent version from the repository.

## Custom Recipe has Stopped Working

**Problem:** A recipe that was working correctly now fails.

**Cause:** AWS OpsWorks recently migrated from Chef 0.9 to Chef 11.4, which introduces some breaking changes.

**Solution:** For more information on the Chef 11 migration, see [Chef Versions \(p. 161\)](#). For more information on Chef 11 changes, see [What's New in Chef 11.0](#).

## Opscode Community Cookbook Does Not Work with AWS OpsWorks

**Problem:** A recipe from an Opscode community cookbook does not work with AWS OpsWorks.

**Cause:** Recipes from Opscode community cookbooks are sometimes not fully compatible with AWS OpsWorks. In particular, AWS OpsWorks does not support search or data bags.

**Solution:** For guidance on modifying Opscode community cookbooks to work with AWS OpsWorks, see [Implementing Recipes for Chef 11.4 Stacks \(p. 162\)](#).

## Overridden Built-in Recipes Don't Run

**Problem:** You override a built-in recipe by creating a private copy and modify it suit your requirements, but it doesn't run.

**Cause:** You can't override built-in recipes. AWS OpsWorks always runs the built-in recipes first, followed by custom recipes. If a custom recipe has the same name as a built-in recipe, Chef assumes that it is a duplicate and does not run it. You can change the recipe name, but it still runs after the built-in recipe.

In addition, note that with Chef 11, including a built-in recipe in a custom recipe will also cause the built-in recipe to fail with a `ResourceNotFound` error. For more information, see [Chef Returns ResourceNotFound \(p. 348\)](#).

**Solution:** You can often achieve the intended result by overriding attributes or templates. For more information, see [Customizing AWS OpsWorks \(p. 230\)](#).

## Troubleshooting AWS OpsWorks

This section contains some commonly encountered AWS OpsWorks issues and their solutions.

### Topics

- [A Layer's Instances All Fail an Elastic Load Balancing Health Check \(p. 349\)](#)
- [Can't Communicate with an Elastic Load Balancing Load Balancer \(p. 349\)](#)
- [An EBS Volume Does Not Reattach After a Reboot \(p. 350\)](#)
- [Can't Delete AWS OpsWorks Security Groups \(p. 350\)](#)
- [Accidentally Deleted an AWS OpsWorks Security Group \(p. 351\)](#)
- [Chef Log Terminates Abruptly \(p. 351\)](#)
- [Custom Cookbook Does Not Get Updated \(p. 351\)](#)
- [Instances are Stuck at Booting Status \(p. 351\)](#)
- [Instances Unexpectedly Restart \(p. 351\)](#)
- [opsworks-agent Processes are Running on Instances \(p. 352\)](#)
- [Unexpected execute\\_recipes commands \(p. 352\)](#)

## A Layer's Instances All Fail an Elastic Load Balancing Health Check

**Problem:** You attach an Elastic Load Balancing load balancer to an app server layer, but all the instances fail the health check.

**Cause:** When you create an Elastic Load Balancing load balancer, you must specify the ping path that the load balancer calls to determine whether the instance is healthy. Be sure to specify a ping path that is appropriate for your application; the default value is `/index.html`. If your application does not include an `index.html`, you must specify an appropriate path or the health check will fail. For example, the SimplePHPApp application used in [Getting Started: Create a Simple PHP Application Server Stack \(p. 9\)](#) does not use `index.html`; the appropriate ping path for those servers is `/`.

**Solution:** Edit the load balancer's ping path. For more information, see [Elastic Load Balancing](#)

## Can't Communicate with an Elastic Load Balancing Load Balancer

**Problem:** You create an Elastic Load Balancing load balancer and attach it to an app server layer, but when you click the load balancer's DNS name or IP address to run the application, you get the following error: "The remote server is not responding".

**Cause:** If your stack is running in a default VPC, when you create an Elastic Load Balancing load balancer in the region, you must specify a security group. The security group must have ingress rules that allow

inbound traffic from your IP address. If you specify **default VPC security group**, the default ingress rule does not accept any inbound traffic.

**Solution:** Edit the security group ingress rules to accept inbound traffic from appropriate IP addresses.

1. Click **Security Groups** in the [Amazon EC2 console's](#) navigation pane.
2. Select the load balancer's security group.
3. Click **Edit** on the **Inbound** tab.
4. Add an ingress rule with **Source** set to an appropriate CIDR.

For example, specifying **Anywhere** sets the CIDR to 0.0.0.0/0, which directs the load balancer to accept incoming traffic from any IP address.

## An EBS Volume Does Not Reattach After a Reboot

**Problem:** You use the Amazon EC2 console to attach an Amazon EBS volume to an instance but when you reboot the instance, the volume is no longer attached.

**Cause:** AWS OpsWorks can reattach only those Amazon EBS volumes that it is aware of, which are limited to the following:

- Volumes that were created by AWS OpsWorks.
- Volumes from your account that you have explicitly registered with a stack by using the [Resources](#) page.

**Solution:** Manage your Amazon EBS volumes only by using the AWS OpsWorks console, API, or CLI. If you want to use one of your account's Amazon EBS volumes with a stack, use the stack's [Resources](#) page to register the volume and attach it to an instance. For more information, see [Resource Management \(p. 218\)](#).

## Can't Delete AWS OpsWorks Security Groups

**Problem:** After you delete a stack, there are a number of AWS OpsWorks security groups left behind that can't be deleted.

**Cause:** The security groups must be deleted in a particular order.

**Solution:** First, make sure that no instances are using the security groups. Then, delete the security groups in the following order:

1. AWS-OpsWorks-Blank-Server
2. AWS-OpsWorks-Monitoring-Master-Server
3. AWS-OpsWorks-DB-Master-Server
4. AWS-OpsWorks-Memcached-Server
5. AWS-OpsWorks-Custom-Server
6. AWS-OpsWorks-nodejs-App-Server
7. AWS-OpsWorks-PHP-App-Server
8. AWS-OpsWorks-Rails-App-Server
9. AWS-OpsWorks-Web-Server
10. AWS-OpsWorks-Default-Server
11. AWS-OpsWorks-LB-Server

## Accidentally Deleted an AWS OpsWorks Security Group

**Problem:** You deleted one of the AWS OpsWorks security groups and need to recreate it.

**Cause:** These security groups are usually deleted by accident.

**Solution:** The recreated group must be an exact duplicate of the original, including the same capitalization for the group name. Instead of recreating the group manually, the preferred approach is to have AWS OpsWorks perform the task for you. Just create a new stack in the same AWS region—and VPC, if present—and AWS OpsWorks will automatically recreate all the built-in security groups, including the one that you deleted. You can then delete the stack if you don't have any further use for it; the security groups will remain.

## Chef Log Terminates Abruptly

**Problem:** A Chef log terminates abruptly; the end of the log does not indicate a successful run or display an exception and stack trace.

**Cause:** This behavior is typically caused by inadequate memory.

**Solution:** Create a larger instance and use the agent CLI `run_command` command to run the recipes again. For more information, see [Executing Recipes \(p. 344\)](#).

## Custom Cookbook Does Not Get Updated

**Problem:** You updated your custom cookbooks but the stack's instances are still running the old recipes.

**Cause:** AWS OpsWorks caches cookbooks on each instance, and runs recipes from the cache, not the repository. When you start a new instance, AWS OpsWorks downloads your custom cookbooks from the repository to the instance's cache. However, if you subsequently modify your custom cookbooks, AWS OpsWorks does not automatically update the online instances' caches.

**Solution:** Run the [Update Cookbooks stack command \(p. 52\)](#) to explicitly direct AWS OpsWorks to update your online instances' cookbook caches.

## Instances are Stuck at Booting Status

**Problem:** When you restart an instance, or auto healing restarts it automatically, the startup operation stalls at the booting status.

**Cause:** One possible cause of this issue is the VPC configuration, including a default VPC. Instances must always be able to communicate with the AWS OpsWorks service, Amazon S3, and the package, cookbook, and app repositories. If, for example, you remove a default gateway from a default VPC, the instances lose their connection to the AWS OpsWorks service. Because AWS OpsWorks can no longer communicate with the instance [agent \(p. 344\)](#), it treats the instance as failed and [auto heals \(p. 67\)](#) it. However, without a connection, AWS OpsWorks cannot install an instance agent on the healed instance. Without an agent, AWS OpsWorks cannot run the Setup recipes on the instance, so the startup operation cannot progress beyond the "booting" status.

**Solution:** Modify your VPC configuration so that instances have the required connectivity. For more information, see [Running a Stack in a VPC \(p. 44\)](#).

## Instances Unexpectedly Restart

**Problem:** A stopped instance unexpectedly restarts.

**Cause 1:** If you have enabled [auto healing \(p. 67\)](#) for your instances, AWS OpsWorks periodically performs a health check on the associated Amazon EC2 instances, and restarts any that are unhealthy. If you stop or terminate an AWS OpsWorks-managed instance by using the Amazon EC2 console, API, or CLI, AWS OpsWorks is not notified. Instead, it will perceive the stopped instance as unhealthy and automatically restart it.

**Solution:** Manage your instances only by using the AWS OpsWorks console, API, or CLI. If you use AWS OpsWorks to stop or delete an instance, it will not be restarted. For more information, see [Manually Starting, Stopping, and Rebooting 24/7 Instances \(p. 109\)](#) and [Deleting Instances \(p. 112\)](#).

**Cause 2:** Instances can fail for a variety of reasons. If you have auto healing enabled, AWS OpsWorks automatically restarts failed instances.

**Solution:** This is normal operation; there is no need to do anything unless you do not want AWS OpsWorks to restart failed instances. In that case, you should disable auto healing.

## opsworks-agent Processes are Running on Instances

**Problem:** Several `opsworks-agent` processes are running on your instances. For example:

```
aws 24543 0.0 1.3 172360 53332 ? S Feb24 0:29 opsworks-agent: master 24543
aws 24545 0.1 2.0 208932 79224 ? S Feb24 22:02 opsworks-agent: keep_alive of
master 24543
aws 24557 0.0 2.0 209012 79412 ? S Feb24 8:04 opsworks-agent: statistics of
master 24543
aws 24559 0.0 2.2 216604 86992 ? S Feb24 4:14 opsworks-agent: process_command
of master 24
```

**Cause:** These are legitimate processes that are required for the agent's normal operation. They perform tasks such as handling deployments and sending keep-alive messages back to the service.

**Solution:** This is normal behavior. Do not stop these processes; doing so will compromise the agent's operation.

## Unexpected execute\_recipes commands

**Problem:** The **Logs** section on an instance's details page includes unexpected `execute_recipes` commands. Unexpected `execute_recipes` commands can also appear on the **Stack** and **Deployments** pages.

**Cause:** This issue is often caused by permission changes. When you change a user or group's SSH or sudo permissions, AWS OpsWorks runs `execute_recipes` to update the stack's instances and also triggers a Configure event. Another source of `execute_recipes` commands is AWS OpsWorks updating the instance agent.

**Solution:** This is normal operation; there is no need to do anything.

To see what actions an `execute_recipes` command performed, go to the **Deployments** page and click the command's time stamp. This opens the command's details page, which lists the key recipes that were run. For example, the following details page is for an `execute_recipes` command that ran `ssh_users` to update SSH permissions.

## Ran command `execute_recipes`

[Repeat](#)

Status	successful	User	OpsWorks
Created at	2014-02-21 17:15:40 UTC	Recipes	ssh_users
Completed at	2014-02-21 17:16:32 UTC		
Duration	00:00:52		
<hr/>			
Hostname	SSH	Layers	Duration
✓ <a href="#">php-app1</a>	PHP App Server	00:00:52	<a href="#">show</a>

To see all of the details, click **show** in the command's **Log** column to display the associated Chef log. Search the log for [Run List](#). AWS OpsWorks maintenance recipes will be under [OpsWorks Custom Run List](#). For example, the following is the run list for the `execute_recipes` command shown in the preceding screenshot, and shows every recipe that is associated with the command.

```
[2014-02-21T17:16:30+00:00] INFO: OpsWorks Custom Run List:  
[ "opsworks_stack_state_sync",  
  "ssh_users", "test_suite", "opsworks_cleanup"]
```

# Appendix A: AWS OpsWorks Layer Reference

---

Every instance that AWS OpsWorks deploys must be a member of at least one layer, which defines an instance's role in the stack and controls the details of setting up and configuring the instance, installing packages, deploying applications, and so on. For more information about how to use the AWS OpsWorks to create and manage layers, see [Layers \(p. 57\)](#).

Each layer description includes a list of the built-in recipes that AWS OpsWorks runs for each of the layer's lifecycle events. Those recipes are stored at <https://github.com/aws/opsworks-cookbooks>. Note that the lists include only those recipes that are run directly by AWS OpsWorks. Those recipes sometimes run dependent recipes, which are not listed. To see the complete list of recipes for a particular event, including dependent and custom recipes, examine the run list in the appropriate [lifecycle event's Chef log \(p. 338\)](#).

## Topics

- [HAProxy Layer Reference \(p. 354\)](#)
- [MySQL Layer Reference \(p. 356\)](#)
- [Web Server Layers Reference \(p. 357\)](#)
- [Custom Layer Reference \(p. 363\)](#)
- [Other Layers \(p. 364\)](#)

## HAProxy Layer Reference

A HAProxy layer uses [HAProxy](#)—a reliable high-performance TCP/HTTP load balancer—to provide high availability load balancing and proxy services for TCP- and HTTP-based applications. It is particularly useful for websites that must crawl under very high loads while requiring persistence or Layer 7 processing.

HAProxy monitors traffic and displays the statistics and the health of the associated instances on a web page. By default, the URI is `http://DNSName/haproxy?stats`, where *DNSName* is the HAProxy instance's DNS name.

**Short name:** lb

**Compatibility:** A HAProxy layer is compatible with the following layers: custom, db-master, and memcached.

**Open ports:** HAProxy allows public access to ports 22 (SSH), 80 (HTTP), and 443 (HTTPS).

**Autoassign Elastic IP addresses:** On by default

**Default EBS volume:** No

**Default security group:** AWS-OpsWorks-LB-Server

**Configuration:** To configure a HAProxy layer, you must specify the following:

- Health check URI (default: http://*DNSName*/).
- Statistics URI (default: http://*DNSName*/haproxy?stats).
- Statistics password (optional).
- Health check method (optional). By default, HAProxy uses the HTTP OPTIONS method. You can also specify GET or HEAD.
- Enable statistics (optional)
- Ports. By default, AWS OpsWorks configures HAProxy to handle both HTTP and HTTPS traffic. You can configure HAProxy to handle only one or the other by overriding the Chef configuration [template](#), `haproxy.cfg.erb`.

**Setup recipes:**

- `opsworks_initial_setup`
- `ssh_host_keys`
- `ssh_users`
- `mysql::client`
- `dependencies`
- `ebs`
- `opsworks_ganglia::client`
- `haproxy`

**Configure recipes:**

- `opsworks_ganglia::configure-client`
- `ssh_users`
- `agent_version`
- `haproxy::configure`

**Deploy recipes:**

- `deploy::default`
- `haproxy::configure`

**Shutdown recipes:**

- `opsworks_shutdown::default`
- `haproxy::stop`

**Installation:**

- AWS OpsWorks uses the instance's package installer to install HAProxy to its default locations.

- You must set up syslog to direct the log files to a specified location. For more information, see [HAProxy](#).

# MySQL Layer Reference

The MySQL layer supports MySQL, a widely used relational database management system. AWS OpsWorks installs the most recent available version, which depends on the operating system. If you add a MySQL instance, the needed access information is provided to the application server layers. You must write custom Chef recipes to set up master–master or master–slave configurations.

**Short name:** db-master

**Compatibility:** A MySQL layer is compatible with the following layers: custom, lb, memcached, monitoring–master, nodejs-app, php-app, rails-app, and web.

**Open ports:** A MySQL layer allows public access to port 22(SSH) and all ports from the stack's web servers, custom servers, and Rails, PHP, and Node.js application servers.

**Autoassign Elastic IP addresses:** Off by default

**Default EBS volume:** Yes, at /vol/mysql

**Default security group:** AWS-OpsWorks-DB-Master-Server

**Configuration:** To configure a MySQL layer, you must specify the following:

- Root user password
- MySQL engine

**Setup recipes:**

- opsworks\_initial\_setup
- ssh\_host\_keys
- ssh\_users
- mysql::client
- dependencies
- ebs
- opsworks\_ganglia::client
- mysql::server
- dependencies
- deploy::mysql

**Configure recipes:**

- opsworks\_ganglia::configure-client
- ssh\_users
- agent\_version
- deploy::mysql

**Deploy recipes:**

- deploy::default
- deploy::mysql

**Shutdown recipes:**

- opsworks\_shutdown::default
- mysql::stop

**Installation:**

- AWS OpsWorks uses the instance's package installer to install MySQL and its log files to their default locations. For more information, see [MySQL Documentation](#).

## Web Server Layers Reference

AWS OpsWorks supports several different application and static web page servers.

**Topics**

- [Java App Server Layer Reference \(p. 357\)](#)
- [Node.js App Server Layer Reference \(p. 358\)](#)
- [PHP App Server Layer Reference \(p. 360\)](#)
- [Rails App Server Layer Reference \(p. 361\)](#)
- [Static Web Server Layer Reference \(p. 362\)](#)

## Java App Server Layer Reference

A Java App Server layer supports an [Apache Tomcat 7.0](#) application server.

**Short name:** java-app

**Compatibility:** A Java App Server layer is compatible with the following layers: custom, db-master, and memcached.

**Open ports:** A Java App Server layer allows public access to ports 22 (SSH), 80 (HTTP), 443 (HTTPS), and all ports from load balancers.

**Autoassign Elastic IP addresses:** Off by default

**Default EBS Volume:** No

**Default security group:** AWS-OpsWorks-Java-App-Server

**Setup recipes:**

- opsworks\_initial\_setup
- ssh\_host\_keys
- ssh\_users
- mysql::client
- dependencies
- ebs
- opsworks\_ganglia::client
- opsworks\_java::setup

**Configure recipes:**

- opsworks\_ganglia::configure-client
- ssh\_users
- agent\_version
- opsworks\_java::configure

**Deploy recipes:**

- deploy::default
- deploy::java

**Undeploy recipes:**

- deploy::java-undeploy

**Shutdown recipes:**

- opsworks\_shutdown::default
- deploy::java-stop

**Installation:**

- Tomcat installs to /usr/share/tomcat7.
- For more information about how to produce log files, see [Logging in Tomcat](#).

## Node.js App Server Layer Reference

A Node.js App Server layer supports a [Node.js](#) application server, which is a platform for implementing highly scalable network application servers. Programs are written in JavaScript, using event-driven asynchronous I/O to minimize overhead and maximize scalability.

**Short name:** nodejs-app

**Compatibility:** A Node.js App Server layer is compatible with the following layers: custom, db-master, memcached, and monitoring-master.

**Open ports:** A Node.js App Server layer allows public access to ports 22 (SSH), 80 (HTTP), 443 (HTTPS), and all ports from load balancers.

**Autoassign Elastic IP addresses:** Off by default

**Default EBS volume:** No

**Default security group:** AWS-OpsWorks-nodejs-App-Server

**Setup recipes:**

- opsworks\_initial\_setup
- ssh\_host\_keys
- ssh\_users
- mysql::client
- dependencies
- ebs

- opsworks\_ganglia::client
- opsworks\_nodejs
- opsworks\_nodejs::npm

**Configure recipes:**

- opsworks\_ganglia::configure-client
- ssh\_users
- agent\_version
- opsworks\_nodejs::configure

**Deploy recipes:**

- deploy::default
- opsworks\_nodejs
- opsworks\_nodejs::npm
- deploy::nodejs

**Undeploy recipes:**

- deploy::nodejs-undeploy

**Shutdown recipes:**

- opsworks\_shutdown::default
- deploy::nodejs-stop

**Installation:**

- Node.js installs to /usr/local/bin/node.
- For more information about how to produce log files, see [How to log in node.js](#).

**Node.js application configuration:**

- The main file run by Node.js must be named `server.js` and reside in the root directory of the deployed application.
- The Node.js application must be set to listen on port 80 (or port 443, if applicable).

**Note**

Node.js apps that run Express commonly use the following code to set the listening port, where `process.env.PORT` represents the default port and resolves to 80:

```
app.set('port', process.env.PORT || 3000);
```

With AWS OpsWorks, you must explicitly specify port 80, as follows:

```
app.set('port', 80);
```

## PHP App Server Layer Reference

The PHP App Server layer supports a PHP application server by using [Apache2](#) with mod\_php.

**Short name:** php-app

**Compatibility:** A PHP App Server layer is compatible with the following layers: custom, db-master, memcached, monitoring-master, and rails-app.

**Open ports:** A PHP App Server layer allows public access to ports 22 (SSH), 80 (HTTP), 443 (HTTPS), and all ports from load balancers.

**Autoassign Elastic IP addresses:** Off by default

**Default EBS volume:** No

**Default security group:** AWS-OpsWorks-PHP-App-Server

**Setup recipes:**

- opsworks\_initial\_setup
- ssh\_host\_keys
- ssh\_users
- mysql::client
- dependencies
- ebs
- opsworks\_ganglia::client
- mysql::client
- dependencies
- mod\_php5\_apache2

**Configure recipes:**

- opsworks\_ganglia::configure-client
- ssh\_users
- agent\_version
- mod\_php5\_apache2::php
- php::configure

**Deploy recipes:**

- deploy::default
- deploy::php

**Undeploy recipes:**

- deploy::php-undeploy

**Shutdown recipes:**

- opsworks\_shutdown::default
- apache2::stop

**Installation:**

- AWS OpsWorks uses the instance's package installer to install Apache2, mod\_php and the associated log files to their default locations. For more information about installation, see [Apache](#). For more information about logging, see [Log Files](#).

## Rails App Server Layer Reference

The Rails App Server layer supports a [Ruby on Rails](#) application server.

**Short name:** rails-app

**Compatibility:** A Rails App Server layer is compatible with the following layers: custom, db-master, memcached, monitoring-master, php-app.

**Ports:** A Rails App Server layer allows public access to ports 22(SSH), 80 (HTTP), 443 (HTTPS), and all ports from load balancers.

**Autoassign Elastic IP addresses:** Off by default

**Default EBS volume:** No

**Default security group:** AWS-OpsWorks-Rails-App-Server

**Configuration:** To configure a Rails App Server layer, you must specify the following:

- Ruby version
- Rails stack
- Rubygems version
- Whether to install and manage [Bundler](#)
- The Bundler version

**Setup recipes:**

- opsworks\_initial\_setup
- ssh\_host\_keys
- ssh\_users
- mysql::client
- dependencies
- ebs
- opsworks\_ganglia::client
- apache2 apache2::mod\_deflate
- passenger\_apache2
- passenger\_apache2::mod\_rails
- passenger\_apache2::rails

**Configure recipes:**

- opsworks\_ganglia::configure-client
- ssh\_users
- agent\_version
- rails::configure

**Deploy recipes:**

- deploy::default
- deploy::rails

**Undeploy recipes:**

- deploy::rails-undeploy

**Shutdown recipes:**

- opsworks\_shutdown::default
- apache2::stop

**Installation:**

- AWS OpsWorks uses the instance's package installer to install Apache2 with mod\_passenger, mod\_rails, and the associated log files to their default locations. For more information about installation, see [Phusion Passenger](#). For more information about logging, see [Log Files](#).

## Static Web Server Layer Reference

The Static Web Server layer serves static HTML pages, which can include client-side code such as JavaScript. It is based on [Nginx](#), which is an open source HTTP, reverse proxy, and mail proxy server.

**Short name:** web

**Compatibility:** A Static Web Server layer is compatible with the following layers: custom, db-master, memcached.

**Open ports:** A Static Web Server layer allows public access to ports 22(SSH), 80 (HTTP), 443 (HTTPS), and all ports from load balancers.

**Autoassign Elastic IP addresses:** Off by default

**Default EBS volume:** No

**Default security group:** AWS-OpsWorks-Web-Server

**Setup recipes:**

- opsworks\_initial\_setup
- ssh\_host\_keys
- ssh\_users
- mysql::client
- dependencies
- ebs
- opsworks\_ganglia::client
- nginx

**Configure recipes:**

- opsworks\_ganglia::configure-client

- ssh\_users
- agent\_version

**Deploy recipes:**

- deploy::default
- deploy::web

**Undeploy recipes:**

- deploy::web-undeploy

**Shutdown recipes:**

- opsworks\_shutdown::default
- nginx::stop

**Installation:**

- Nginx installs to /usr/sbin/nginx.
- Nginx log files are in /var/log/nginx.

## Custom Layer Reference

If the standard layers don't suit your requirements, you can create a custom layer. A stack can have multiple custom layers. By default, the custom layer runs a limited set of standard recipes that support basic functionality, such as Ganglia monitoring. You then implement the layer's primary functionality by implementing a set of custom Chef recipes for each of the appropriate lifecycle events to set up and configure the layer's software, and so on. Custom recipes run after the standard AWS OpsWorks recipes for each event.

**Short name:** User-defined; each custom layer in a stack must have a different short name

**Open ports:** By default, a custom server layer opens public access to ports 22(SSH), 80 (HTTP), 443 (HTTPS), and all ports from the stack's Rails and PHP application server layers

**Autoassign Elastic IP Addresses:** Off by default

**Default EBS volume:** No

**Default Security Group:** AWS-OpsWorks-Custom-Server

**Compatibility:** Custom layers are compatible with the following layers: custom, db-master, lb, memcached, monitoring-master, nodejs-app, php-app, rails-app, and web

**Configuration:** To configure a custom layer, you must specify the following:

- The layer's name
- The layer's short name, which identifies the layer in Chef recipes and must use only a-z and numbers

**Setup recipes:**

- opsworks\_initial\_setup

- ssh\_host\_keys
- ssh\_users
- mysql::client
- dependencies
- ebs
- opsworks\_ganglia::client

**Configure recipes:**

- opsworks\_ganglia::configure-client
- ssh\_users
- agent\_version

**Deploy recipes:**

- deploy::default

**Shutdown recipes:**

- opsworks\_shutdown::default

## Other Layers

AWS OpsWorks also supports the following layers.

**Topics**

- [Ganglia Layer Reference \(p. 364\)](#)
- [Memcached Layer Reference \(p. 366\)](#)

## Ganglia Layer Reference

A Ganglia layer supports [Ganglia](#), a distributed monitoring system that manages the storage and visualization of instance metrics. It is designed to work with hierarchical instance topologies, which makes it particularly useful for groups of instances. Ganglia has two basic components:

- A low-overhead client, which is installed on each instance in the stack and sends metrics to the master.
- A master, which collects metrics from the clients and stores them on an Amazon EBS volume. It also displays the metrics on a web page.

AWS OpsWorks has a Ganglia monitoring agent on each instance that it manages. When you add a Ganglia layer to your stack and start it, the Ganglia agents on each instance report metrics to the Ganglia instance. To use Ganglia, add a Ganglia layer with one instance to the stack. You access the data by logging in to the Ganglia backend at the master's IP address. You can provide additional metric definitions by writing Chef recipes.

**Short name:** monitoring-master

**Compatibility:** A Ganglia layer is compatible with the following layers: custom, db-master, memcached, php-app, rails-app.

**Open ports:** Load-Balancer allows public access to ports 22(SSH), 80 (HTTP), and 443 (HTTPS).

**Autoassign Elastic IP addresses:** Off by default

**Default EBS volume:** Yes, at `/vol/ganglia`

**Default security group:** AWS-OpsWorks-Monitoring-Master-Server

**Configuration:** To configure a Ganglia layer, you must specify the following:

- The URI that provides access to the monitoring graphs. The default value is `http://DNSName/ganglia`, where *DNSName* is the Ganglia instance's DNS name.
- A user name and password that control access to the monitoring statistics.

**Setup recipes:**

- `opsworks_initial_setup`
- `ssh_host_keys`
- `ssh_users`
- `mysql::client`
- `dependencies`
- `ebs`
- `opsworks_ganglia::client`
- `opsworks_ganglia::server`

**Configure recipes:**

- `opsworks_ganglia::configure-client`
- `ssh_users`
- `agent_version`
- `opsworks_ganglia::configure-server`

**Deploy recipes:**

- `deploy::default`
- `opsworks_ganglia::configure-server`
- `opsworks_ganglia::deploy`

**Shutdown recipes:**

- `opsworks_shutdown::default`
- `apache2::stop`

**Installation:**

- The Ganglia client is installed under: `/etc/ganglia`.
- The Ganglia web front end is installed under: `/usr/share/ganglia-webfrontend`.
- The Ganglia logtailer is installed under: `/usr/share/ganglia-logtailer`.

## Memcached Layer Reference

**Memcached** is a distributed memory-caching system for arbitrary data. It speeds up websites by caching strings and objects as keys and values in RAM to reduce the number of times an external data source must be read.

To use Memcached in a stack, create a Memcached layer and add one or more instances, which function as Memcached servers. The instances automatically install Memcached and the stack's other instances are able to access and use the Memcached servers. If you use a Rails App Server layer, AWS OpsWorks automatically places a `memcached.yml` configuration file in the config directory of each instance in the layer. You can obtain the Memcached server and port number from this file.

**Short name:** memcached

**Compatibility:** A Memcached layer is compatible with the following layers: custom, db-master, lb, monitoring-master, nodejs-app, php-app, rails-app, and web.

**Open ports:** A Memcached layer allows public access to port 22(SSH) and all ports from the stack's web servers, custom servers, and Rails, PHP, and Node.js application servers.

**Autoassign Elastic IP addresses:** Off by default

**Default EBS volume:** No

**Default security group:** AWS-OpsWorks-Memcached-Server

To configure a Memcached layer, you must specify the cache size, in MB.

**Setup recipes:**

- `opsworks_initial_setup`
- `ssh_host_keys`
- `ssh_users`
- `mysql::client`
- `dependencies`
- `ebs`
- `opsworks_ganglia::client`
- `memcached`

**Configure recipes:**

- `opsworks_ganglia::configure-client`
- `ssh_users`
- `agent_version`

**Deploy recipes:**

- `deploy::default`

**Shutdown recipes:**

- `opsworks_shutdown::default`
- `memcached::stop`

**Installation:**

- AWS OpsWorks uses the instance's package installer to install Memcached and its log files in their default locations.

# Appendix B: Instance Agent CLI

---

The instance agent that AWS OpsWorks installs on every instance exposes a command line interface (CLI). If you connect using SSH to the instance, you can use the CLI to the following:

- Access log files for Chef runs
- Access AWS OpsWorks commands
- Manually run Chef recipes
- View instance reports
- View agent reports
- View the [stack configuration and deployment JSON \(p. 262\)](#)

### Important

You can run agent CLI commands only as root or by using `sudo`.

The basic command syntax is:

```
sudo opsworks-agent-cli [--help] [command [activity] [date]]
```

The four arguments are as follows:

#### **help**

(Optional) Displays a brief synopsis of the available commands when used by itself. When used with a command, `help` displays a description of the command.

#### **command**

(Optional) The agent CLI command, which must be set to one of the following:

- [agent\\_report \(p. 369\)](#)
- [get\\_json \(p. 369\)](#)
- [instance\\_report \(p. 372\)](#)
- [list\\_commands \(p. 373\)](#)
- [run\\_command \(p. 373\)](#)
- [show\\_log \(p. 374\)](#)
- [stack\\_state \(p. 375\)](#)

#### **activity**

(Optional) Used as an argument with some commands to specify a particular AWS OpsWorks activity: `setup`, `configure`, `deploy`, `undeploy`, `start`, `stop`, or `restart`.

**date**

(Optional) Used as an argument with some commands to specify a particular AWS OpsWorks command execution. Specify the command execution by setting **date** to the time that the command was executed in the 'yyyy-mm-ddd-hh-mm-ss' format, including the single quotes. For example, for 10:31:55 on Tuesday Feb 5, 2013, use: '2013-02-05T10:31:55'. To determine when a particular AWS OpsWorks command was executed, run [list\\_commands \(p. 373\)](#).

**Note**

If the agent has executed the same AWS OpsWorks activity multiple times, you can pick a particular execution by specifying both the activity and the time it was executed. If you specify an activity and omit the time, the agent CLI command acts on the activity's most recent execution. If you omit both arguments, the agent CLI command acts on the most recent activity.

The following sections describe the commands and their associated arguments. For brevity, the syntax sections omit the optional --help option, which can be used with any command.

**Topics**

- [agent\\_report \(p. 369\)](#)
- [get\\_json \(p. 369\)](#)
- [instance\\_report \(p. 372\)](#)
- [list\\_commands \(p. 373\)](#)
- [run\\_command \(p. 373\)](#)
- [show\\_log \(p. 374\)](#)
- [stack\\_state \(p. 375\)](#)

## agent\_report

Displays an agent report.

```
sudo opsworks-agent-cli agent_report
```

The following output example is from a PHP App Server instance that most recently ran a deploy activity.

```
[ec2-user@php-app1 ~]$ sudo opsworks-agent-cli agent_report

AWS OpsWorks Instance Agent State Report:

Last activity was a "deploy" on Tue Feb 26 19:21:13 UTC 2013
Agent Status: The AWS OpsWorks agent is running as PID 1282
Agent Version: 104, up to date
```

## get\_json

Displays a command's stack configuration JSON.

```
sudo opsworks-agent-cli get_json [activity] [date]
```

By default, `get_json` displays the stack configuration JSON for the most recent activity. Use the following options to specify a particular file.

**activity**

Displays the specified activity's JSON file.

**date**

Displays the JSON file for the activity that executed at the specified date.

The following output example is from a PHP App Server instance and shows the JSON for the most recent `configure` activity.

```
[ec2-user@php-app1 ~]$ sudo opsworks-agent-cli get_json configure
{
    "opsworks": {
        "layers": {
            "php-app": {
                "id": "4a2a56c8-f909-4b39-81f8-556536d20648",
                "instances": {
                    "php-app2": {
                        "elastic_ip": null,
                        "region": "us-west-2",
                        "booted_at": "2013-02-26T20:41:10+00:00",
                        "ip": "10.112.235.192",
                        "aws_instance_id": "i-34037f06",
                        "availability_zone": "us-west-2a",
                        "instance_type": "c1.medium",
                        "private_dns_name": "ip-10-252-0-203.us-west-2.compute.internal",
                        "private_ip": "10.252.0.203",
                        "created_at": "2013-02-26T20:39:39+00:00",
                        "status": "online",
                        "backends": 8,
                        "public_dns_name": "ec2-10-112-235-192.us-west-2.compute.amazonaws.com"
                    },
                    "php-app1": {
                        "elastic_ip": null,
                        "region": "us-west-2",
                        "booted_at": "2013-02-26T20:41:09+00:00",
                        "ip": "10.112.44.160",
                        "aws_instance_id": "i-32037f00",
                        "availability_zone": "us-west-2a",
                        "instance_type": "c1.medium",
                        "private_dns_name": "ip-10-252-84-253.us-west-2.compute.internal",
                        "private_ip": "10.252.84.253",
                        "created_at": "2013-02-26T20:39:35+00:00",
                        "status": "online",
                        "backends": 8,
                        "public_dns_name": "ec2-10-112-44-160.us-west-2.compute.amazonaws.com"
                    }
                }
            },
            "name": "PHP Application Server"
        },
        "lb": {
            "id": "15c86142-d836-4191-860f-f4d310440f14",
            "type": "application"
        }
    }
}
```

```
"instances": {
    "lb1": {
        "elastic_ip": "54.244.244.50",
        "region": "us-west-2",
        "booted_at": "2013-02-26T20:41:14+00:00",
        "ip": null,
        "aws_instance_id": "i-3a037f08",
        "availability_zone": "us-west-2a",
        "instance_type": "c1.medium",
        "private_dns_name": "ip-10-253-1-116.us-west-2.compute.internal",
        "private_ip": "10.253.1.116",
        "created_at": "2013-02-26T20:39:44+00:00",
        "status": "online",
        "backends": 8,
        "public_dns_name": null
    }
},
"name": "Load Balancer"
},
"agent_version": "104",
"applications": [
],
"stack": {
    "name": "MyStack"
},
"ruby_version": "1.8.7",
"sent_at": 1361911623,
"ruby_stack": "ruby_enterprise",
"instance": {
    "layers": [
        "php-app"
    ],
    "region": "us-west-2",
    "ip": "10.112.44.160",
    "id": "45ef378d-b87c-42be-alb9-b67c48edaf4",
    "aws_instance_id": "i-32037f00",
    "availability_zone": "us-west-2a",
    "private_dns_name": "ip-10-252-84-253.us-west-2.compute.internal",
    "instance_type": "c1.medium",
    "hostname": "php-app1",
    "private_ip": "10.252.84.253",
    "backends": 8,
    "architecture": "i386",
    "public_dns_name": "ec2-10-112-44-160.us-west-2.compute.amazonaws.com"
},
"activity": "configure",
"rails_stack": {
    "name": null
},
"deployment": null,
"valid_client_activities": [
    "reboot",
    "stop",
    "setup",
    "configure",
    "update_dependencies",
]
```

```
        "install_dependencies",
        "update_custom_cookbooks",
        "execute_recipes"
    ],
},
"opsworks_custom_cookbooks": {
    "recipes": [
        ],
        "enabled": false
    },
    "recipes": [
        "opsworks_custom_cookbooks::load",
        "opsworks_ganglia::configure-client",
        "ssh_users",
        "agent_version",
        "mod_php5_apache2::php",
        "php::configure",
        "opsworks_stack_state_sync",
        "opsworks_custom_cookbooks::execute",
        "test_suite",
        "opsworks_cleanup"
    ],
    "opsworks_rubygems": {
        "version": "1.8.24"
    },
    "ssh_users": {
    },
    "opsworks_bundler": {
        "manage_package": null,
        "version": "1.0.10"
    },
    "deploy": {
    }
}
```

## instance\_report

Displays an extended instance report.

```
sudo opsworks-agent-cli instance_report
```

The following output example is from a PHP App Server instance.

```
[ec2-user@php-app1 ~]$ sudo opsworks-agent-cli instance_report

AWS OpsWorks Instace Agent State Report:

Last activity was a "deploy" on Tue Feb 26 19:21:13 UTC 2013
Agent Status: The AWS OpsWorks agent is running as PID 1282
Agent Version: 104, up to date
OpsWorks Stack: MyStack
OpsWorks Layers: php-app
```

```
OpsWorks Instance: php-app1
EC2 Instace ID: i-065fc875
EC2 Instance Type: c1.medium
Architecture: i386
Total Memory: 1.76 Gb
CPU: 2x Intel(R) Xeon(R) CPU E5410 @ 2.33GHz
```

Location:

```
EC2 Region: us-east-1
EC2 Availability Zone: us-east-1a
```

Networking:

```
Public IP: 204.236.200.220
Private IP: domU-12-31-39-01-96-8A.compute-1.internal
```

## list\_commands

Lists the time for each activity that has been executed on this instance. You can use these times for other agent-CLI commands to specify a particular execution.

```
sudo opsworks-agent-cli list_commands [activity] [date]
```

The following output example is from a PHP App Server instance that has run a setup, deploy, and two configure activities:

```
[ec2-user@php-app1 ~]$ sudo opsworks-agent-cli list_commands
2013-02-26T19:08:26      setup
2013-02-26T19:12:01      configure
2013-02-26T19:12:05      configure
2013-02-26T19:22:12      deploy
```

## run\_command

Runs an AWS OpsWorks command, which is a JSON file containing a Chef run-list that contains the information necessary to execute an AWS OpsWorks activity (setup, configure, deploy, and so on). The `run_command` command generates a log entry that you can view by running [show\\_log \(p. 374\)](#). This option is intended only for development purposes, so AWS OpsWorks does not track changes.

```
sudo opsworks-agent-cli run_command [activity] [date] [/path/to/valid/json.file]
```

By default, `run_command` runs the most recent AWS OpsWorks command. Use the following options to specify a particular command.

### activity

Run a specified AWS OpsWorks command: `setup`, `configure`, `deploy`, `undeploy`, `start`, `stop`, or `restart`.

**date**

Run the AWS OpsWorks command that executed at the specified time. To get a list of times, run [list\\_commands \(p. 373\)](#).

**file**

Run the specified command JSON file. To get a command's file path, run [get\\_json \(p. 369\)](#).

The following output example is from a PHP App Server instance and runs the configure command.

```
[ec2-user@php-app1 ~]$ sudo opsworks-agent-cli run_command configure
[2013-02-26 19:56:58] INFO [opsworks-agent (6694)] About to re-run 'configure'
from 2013-02-26T19:12:05
Waiting for process 6697
pid 6697 exited with status 0, exit code: 0 (time=4 sec)
[2013-02-26 19:57:02] INFO [opsworks-agent (6694)] Finished Chef run with exit
code 0
```

## show\_log

Displays a command's log file.

```
sudo opsworks-agent-cli show_log [activity] [date]
```

By default, `show_log` tails the most recent log file. Use the following options to specify a particular command.

**activity**

Display the specified activity's log file.

**date**

Display the log file for the activity that executed at the specified time.

The following output example is from a PHP App Server instance and shows the most recent log.

```
[ec2-user@php-app1 ~]$ sudo opsworks-agent-cli show_log
[Tue, 26 Feb 2013 19:22:20 +0000] INFO: Chef Run complete in 6.441339 seconds
[Tue, 26 Feb 2013 19:22:20 +0000] INFO: cleaning the checksum cache
[Tue, 26 Feb 2013 19:22:20 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--etc-sudoers
[Tue, 26 Feb 2013 19:22:20 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--tmp-chef-rendered-template20130226-3288-
y75s0q-0
[Tue, 26 Feb 2013 19:22:20 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--tmp-chef-rendered-template20130226-3288-
1jlrfv4-0
[Tue, 26 Feb 2013 19:22:20 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--tmp-chef-rendered-template20130226-3288-
s6k78b-0
[Tue, 26 Feb 2013 19:22:20 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--var-lib-aws-opsworks-TARGET_VERSION
[Tue, 26 Feb 2013 19:22:20 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--tmp-chef-rendered-template20130226-3288-
```

```
1bq1mfg-0
[Tue, 26 Feb 2013 19:22:20 +0000] DEBUG: removing unused checksum cache file
/var/chef/cache/checksums/chef-file--etc-ganglia-gmond-conf
[Tue, 26 Feb 2013 19:22:20 +0000] INFO: Running report handlers
[Tue, 26 Feb 2013 19:22:20 +0000] INFO: Report handlers complete
[Tue, 26 Feb 2013 19:22:20 +0000] DEBUG: Exiting
```

## stack\_state

Displays the current stack configuration JSON.

```
opsworks-agent-cli stack_state
```

The following output example is from a PHP App Server instance.

```
[ec2-user@php-app1 ~]$ sudo opsworks-agent-cli stack_state
{
    "layers": {
        "php-app": {
            "instances": {
                "php-app2": {
                    "availability_zone": "us-east-1a",
                    "instance_type": "c1.medium",
                    "backends": 8,
                    "booted_at": "2013-02-26T19:05:33+00:00",
                    "status": "online",
                    "ip": "10.243.25.193",
                    "region": "us-east-1",
                    "public_dns_name": "ec2-10-243-25-193.compute-1.amazonaws.com",
                    "private_ip": "domU-12-31-39-0C-84-11.compute-1.internal",
                    "elastic_ip": null,
                    "created_at": "2013-02-26T19:03:43+00:00",
                    "aws_instance_id": "i-005fc873",
                    "private_dns_name": "domU-12-31-39-0C-84-11.compute-1.internal"
                },
                "php-app1": {
                    "availability_zone": "us-east-1a",
                    "instance_type": "c1.medium",
                    "backends": 8,
                    "booted_at": "2013-02-26T19:05:36+00:00",
                    "status": "online",
                    "ip": "10.236.200.220",
                    "region": "us-east-1",
                    "public_dns_name": "ec2-10-236-200-220.compute-1.amazonaws.com",
                    "private_ip": "domU-12-31-39-01-96-8A.compute-1.internal",
                    "elastic_ip": null,
                    "created_at": "2013-02-26T19:03:38+00:00",
                    "aws_instance_id": "i-065fc875",
                    "private_dns_name": "domU-12-31-39-01-96-8A.compute-1.internal"
                }
            },
            "id": "0539f6b4-9080-4fc9-a7ae-1c4514f074d2",
            "name": "PHP Application Server"
        },
    }
}
```

```
"lb": {
    "instances": {
        "lb1": {
            "availability_zone": "us-east-1a",
            "instance_type": "c1.medium",
            "backends": 8,
            "booted_at": "2013-02-26T19:06:03+00:00",
            "status": "online",
            "ip": "10.22.247.194",
            "region": "us-east-1",
            "public_dns_name": "ec2-10-22-247-194.compute-1.amazonaws.com",
            "private_ip": "domU-12-31-39-09-08-A8.compute-1.internal",
            "elastic_ip": null,
            "created_at": "2013-02-26T19:03:50+00:00",
            "aws_instance_id": "i-465dca35",
            "private_dns_name": "domU-12-31-39-09-08-A8.compute-1.internal"
        }
    },
    "id": "a20986c1-c9d5-45f0-a565-bb43ddd34150",
    "name": "Load Balancer"
},
"last_command": {
    "sent_at": 1361906473,
    "activity": "deploy"
},
"instance": {
    "availability_zone": "us-east-1a",
    "backends": 8,
    "instance_type": "c1.medium",
    "ip": "10-456-789-012",
    "id": "1a78dd65-a365-4f54-9027-8b6234ee65d4",
    "region": "us-east-1",
    "public_dns_name": "ec2-10-456-789-012.compute-1.amazonaws.com",
    "private_ip": "domU-12-31-39-01-96-8A.compute-1.internal",
    "layers": [
        "php-app"
    ],
    "architecture": "i386",
    "hostname": "php-app1",
    "aws_instance_id": "i-065fc875",
    "private_dns_name": "domU-12-31-39-01-96-8A.compute-1.internal"
},
"stack": {
    "name": "MyStack"
},
"agent": {
    "valid_activities": [
        "reboot",
        "stop",
        "setup",
        "configure",
        "update_dependencies",
        "install_dependencies",
        "update_custom_cookbooks",
        "execute_recipes"
    ]
},
```

```
"applications": [  
    {  
        "slug_name": "simplephpapp",  
        "name": "SimplePHPApp",  
        "application_type": "php"  
    }  
]
```

# Appendix C: AWS OpsWorks Attribute Reference

AWS OpsWorks exposes a wide variety of settings to recipes as Chef attributes. This appendix is a reference for the commonly used attributes.

The attribute definitions derive from two sources:

- The [Stack Configuration and Deployment JSON \(p. 262\)](#).

AWS OpsWorks installs a current version of this JSON on the stack's instances for each lifecycle event.

- The built-in cookbook [attributes files \(p. 156\)](#), which are installed on each instance during setup.

Each attribute has a value, which can be any of the following types:

- **String** values for JSON attributes must be contained in double quotes. Cookbook attributes follow standard Ruby syntax so strings can use single or double quotes, although strings containing certain special characters must have double quotes. For more information, go to the [Ruby](#) documentation site.
- **Boolean** values are either `true` or `false` (no quotes).
- **Number** values are either integer or decimal numbers, such as `4` or `2.5` (no quotes).
- **List** values take the form of comma-separated values enclosed in square brackets (no quotes), such as `[ '80', '443' ]`
- **JSON objects** contain one or more attribute definitions, such as `"php-app2": { "elastic_ip": null, ... }`.

Your custom recipes can access attribute values by using the standard Chef `node[ :attribute ][ :child_attribute ][ :... ]` syntax. For example, the following represents an activity, such as `deploy` or `configure`, that has occurred on an instance.

```
node[ :opsworks ][ :activity ]
```

You can also use quotes rather than colons, as follows:

```
node[ "opsworks" ][ "activity" ]
```

You can override these settings by using custom JSON or custom attributes, as described in [Customizing AWS OpsWorks Configuration by Overriding Attributes \(p. 231\)](#). However, you should be aware that overriding some settings, such as the attributes in the `opsworks` namespace, might interfere with the correct operation of built-in recipes.

For more discussion of stack configuration JSON and recipes, see [Stack Configuration and Deployment JSON \(p. 262\)](#).

This appendix is a reference for the commonly used AWS OpsWorks attributes.

## Stack Configuration and Deployment JSON Attributes

When AWS OpsWorks runs an *activity* on an instance—for example, a setup activity after an instance boots or a deploy activity in response to a deployment—it installs a JSON object on the instance that describes the stack's current configuration. For deployments, the object includes additional deployment-related attributes. Your custom recipes can access these JSON values by using standard Chef `node[:attribute][:sub_attribute][:...]` syntax.

This section includes the most commonly used stack configuration and deployment JSON attributes and their associated node syntax. Note that the same attribute names are sometimes used for different purposes, and occur in different namespaces. For example, `id` can refer to a stack ID, a layer ID, an app ID, and so on, so you need the fully qualified name to use the attribute value. For that reason, this attribute reference is organized around the stack configuration namespace structure. For examples of stack and deployment configuration JSON, see [Stack Configuration and Deployment JSON \(p. 262\)](#).

The following are general guidelines for using stack configuration and deployment JSON attributes:

- Some attributes, such as the ones that define layer configurations, are common to the stack as a whole.

These attributes and their values are the same for JSON associated with any activity, and change only when the stack configuration changes.

- Some attributes depend on the instance.

For example, the `instance_id` value is specific to a particular instance.

- Some of the attributes depend on the activity.

For example, deployment-related attributes are included only for deploy activities.

- Layers and instances are referred to by their short names, such as "1b" and "1b1". Apps are referred to by their slug names, which is a short name such as "`simplephp`" that is generated by AWS OpsWorks from the user-defined name.
- Most attributes use AWS OpsWorks-defined names that are the same for all JSON objects.

Some attributes, such as custom layer names and app names, have user-defined names.

- Attributes aren't always listed in the same order.

### Topics

- [opsworks Attributes \(p. 380\)](#)
- [opsworks\\_custom\\_cookbooks Attributes \(p. 392\)](#)
- [dependencies Attributes \(p. 392\)](#)
- [ganglia Attributes \(p. 392\)](#)
- [mysql Attributes \(p. 393\)](#)

- [passenger Attributes \(p. 393\)](#)
- [opsworks\\_bundler Attributes \(p. 394\)](#)
- [deploy Attributes \(p. 394\)](#)
- [Other Top-Level Attributes \(p. 399\)](#)

## opsworks Attributes

The `opsworks` element—sometimes referred to as the `opsworks` namespace—contains a set of attributes that define the basic stack configuration.

### Important

Overriding the attribute values in the `opsworks` namespace is not recommended. Doing so can cause the built-in recipes to fail.

### Topics

- [applications \(p. 380\)](#)
- [instance Attributes \(p. 381\)](#)
- [layers Attributes \(p. 382\)](#)
- [rails\\_stack Attributes \(p. 385\)](#)
- [stack Attributes \(p. 386\)](#)
- [Other Top-level opsworks Attributes \(p. 391\)](#)

## applications

Contains a list of embedded objects, one for each app that exists for the stack. Each embedded object contains the following attributes that describe the load balancer configuration.

### Note

The general node syntax for these attributes is as follows, where `i` specifies the instance's zero-based list index.

```
node[ :opsworks ][ :applications ][ i ][ :attribute_name ]
```

### application\_type

The application's type (string). Possible values are as follows:

- "php": PHP app
- "rails": A Ruby on Rails app
- "node": A JavaScript app
- "web": A static HTML page
- "other": All other application types

```
node[ :opsworks ][ :applications ][ i ][ :application_type ]
```

### name

The user-defined display name, such as "SimplePHP" (string).

```
node[ :opsworks ][ :applications ][ i ][ :name ]
```

**slug\_name**

A short name, which is an all-lowercase name such as "simplephp" that is generated by OpsWorks from the app's name (string).

```
node[:opsworks][:applications][i][:slug_name]
```

## instance Attributes

The `instance` attribute contains a set of attributes that specify the configuration of this instance.

<a href="#">architecture</a> (p. 381)	<a href="#">availability_zone</a> (p. 381)	<a href="#">backends</a> (p. 381)
<a href="#">aws_instance_id</a> (p. 381)	<a href="#">hostname</a> (p. 381)	<a href="#">id</a> (p. 381)
<a href="#">instance_type</a> (p. 382)	<a href="#">ip</a> (p. 382)	<a href="#">layers</a> (p. 382)
<a href="#">private_dns_name</a> (p. 382)	<a href="#">private_ip</a> (p. 382)	<a href="#">public_dns_name</a> (p. 382)
<a href="#">region</a> (p. 382)		

**architecture**

The instance's architecture, such as "i386" (string).

```
node[:opsworks][:instance][:architecture]
```

**availability\_zone**

The instance's availability zone, such as "us-east-1a" (string).

```
node[:opsworks][:instance][:availability_zone]
```

**backends**

The number of back-end web processes (string). It determines, for example, the number of concurrent connections that HAProxy will forward to a Rails back end. The default value depends on the instance's memory and number of cores.

```
node[:opsworks][:instance][:backends]
```

**aws\_instance\_id**

The EC2 instance ID (string).

```
node[:opsworks][:instance][:aws_instance_id]
```

**hostname**

The host name, such as "php-app1" (string).

```
node[:opsworks][:instance][:hostname]
```

**id**

The instance ID, which is an AWS OpsWorks-generated GUID that uniquely identifies the instance (string).

```
node[:opsworks][:instance][:id]
```

**instance\_type**

The instance type, such as "c1.medium" (string).

```
node[:opsworks][:instance][:instance_type]
```

**ip**

The public IP address (string).

```
node[:opsworks][:instance][:ip]
```

**layers**

A list of the instance's layers, which are identified by their short names, such as "lb" or "db-master" (list of string).

```
node[:opsworks][:instance][:layers]
```

**private\_dns\_name**

The private DNS name (string).

```
node[:opsworks][:instance][:private_dns_name]
```

**private\_ip**

The private IP address (string).

```
node[:opsworks][:instance][:private_ip]
```

**public\_dns\_name**

The public DNS name (string).

```
node[:opsworks][:instance][:public_dns_name]
```

**region**

The AWS region, such as "us-east-1" (string).

```
node[:opsworks][:instance][:region]
```

## layers Attributes

The `layers` attribute contains a set of layer attributes, one for each of the stack's layers, which are named with the layer's short name, such as `php-app`. A stack can have at most one each of the built-in layers, whose short names are as follows:

- `lb`: HAProxy layer
- `db-master`: MySQL layer
- `nodejs-app`: Node.js App Server layer
- `php-app`: PHP App Server layer
- `rails-app`: Rails App Server layer

- **web**: Static Web Server layer
- **monitoring-master**: Ganglia layer
- **memcached**: Memcached layer

A stack can contain any number of custom layers, which have user-defined short names.

Each layer attribute contains the following attributes:

- [id](#) (p. 383)
- [instances](#) (p. 383)
- [name](#) (p. 385)

#### **id**

The layer ID, which is a GUID that is generated by OpsWorks and uniquely identifies the layer (string).

```
node[:opsworks][:layers]['layershortname'][:id]
```

#### **instances**

The `instances` element contains a set of instance attributes, one for each of the layer's online instances. They are named with the instance's short name, such as `php-app1`.

##### **Note**

The `instances` element contains only those instances that are in the online state when the particular stack and deployment JSON is created.

Each instance element contains the following attributes:

<a href="#">availability_zone</a> (p. 383)	<a href="#">aws_instance_id</a> (p. 383)	<a href="#">backends</a> (p. 383)
<a href="#">booted_at</a> (p. 384)	<a href="#">created_at</a> (p. 384)	<a href="#">elastic_ip</a> (p. 384)
<a href="#">instance_type</a> (p. 384)	<a href="#">ip</a> (p. 384)	<a href="#">private_ip</a> (p. 384)
<a href="#">public_dns_name</a> (p. 384)	<a href="#">private_dns_name</a> (p. 384)	<a href="#">region</a> (p. 385)
<a href="#">status</a> (p. 385)		

#### **availability\_zone**

The Availability Zone, such as "us-east-1a" (string).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:availability_zone]
```

#### **aws\_instance\_id**

The EC2 instance ID (string).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:ws_instance_id]
```

#### **backends**

The number of back-end web processes (number). It determines, for example, the number of concurrent connections that HAProxy will forward to a Rails back end. The default value depends on the instance's memory and number of cores.

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:backends]
```

**booted\_at**

The time that the EC2 instance was booted, using the UTC yyyy-mm-ddThh:mm:ss+hh:mm format (string). For example, "2013-10-01T08:35:22+00:00" corresponds to 8:35:22 on Oct. 10, 2013, with no time zone offset. For more information, see [ISO 8601](#).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:booted_at]
```

**created\_at**

The time that the EC2 instance was created, using the UTC yyyy-mm-ddThh:mm:ss+hh:mm format (string). For example, "2013-10-01T08:35:22+00:00" corresponds to 8:35:22 on Oct. 10, 2013, with no time zone offset. For more information, see [ISO 8601](#).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:created_at]
```

**elastic\_ip**

The Elastic IP address, which is set to null if the instance does not have one (string).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:elastic_ip]
```

**instance\_type**

The instance type, such as "c1.medium" (string).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:instance_type]
```

**ip**

The public IP address (string).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:ip]
```

**private\_ip**

The private IP address (string).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:private_ip]
```

**public\_dns\_name**

The public DNS name (string).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][:public_dns_name]
```

**private\_dns\_name**

The private DNS name (string).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][::private_dns_name]
```

**region**

The AWS region, such as "us-east-1" (string).

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][::region]
```

**status**

The status (string). Possible values are as follows:

- "requested"
- "booting"
- "running\_setup"
- "online"
- "setup\_failed"
- "start\_failed"
- "terminating"
- "terminated"
- "stopped"
- "connection\_lost"

```
node[:opsworks][:layers]['layershortname'][:instances]['instanceshortname'][::status]
```

**name**

The layer's name, which is used to represent the layer in the console (string). It can be user-defined and is not necessarily unique.

```
node[:opsworks][:layers]['layershortname'][::name]
```

## rails\_stack Attributes

**name**

Specifies the rails stack, and is set to "apache\_passenger" or "nginx\_unicorn" (string).

```
node[:opsworks][:rails_stack][::name]
```

**recipe**

The associated recipe, which depends on whether you are using Passenger or Unicorn (string):

- Unicorn: "unicorn::rails"
- Passenger: "passenger\_apache2::rails"

```
node[:opsworks][:rails_stack][::recipe]
```

**restart\_command**

The restart command, which depends on whether you are using Passenger or Unicorn (string):

- Unicorn: ".../shared/scripts/unicorn clean-restart"

- Passenger: "touch tmp/restart.txt"

#### service

The service name, which depends on whether you are using Passenger or Unicorn (string):

- Unicorn: "unicorn"
- Passenger: "apache2"

```
node[:opsworks][:rails_stack][:service]
```

## stack Attributes

stack attributes specify some aspects of the stack configuration, such as service layer configurations.

- [elb-load-balancers](#) (p. 386)
- [id](#) (p. 386)
- [name](#) (p. 386)
- [rds\\_instances](#) (p. 387)
- [vpc\\_id](#) (p. 391)

#### elb-load-balancers

Contains a list of embedded objects, one for each Elastic Load Balancing load balancer in the stack. Each embedded object contains the following attributes that describe the load balancer configuration.

##### Note

The general node syntax for these attributes is as follows, where *i* specifies the instance's zero-based list index.

```
node[:opsworks][:stack]['elb-load-balancers'][i][:attribute_name]
```

#### dns\_name

The load balancer's DNS name (string).

```
node[:opsworks][:stack]['elb-load-balancers'][i][:dns_name]
```

#### name

The load balancer's name (string).

```
node[:opsworks][:stack]['elb-load-balancers'][i][:name]
```

#### layer\_id

The ID of the layer that the load balancer is attached to (string).

```
node[:opsworks][:stack]['elb-load-balancers'][i][:layer_id]
```

#### id

The stack ID (string).

```
node[:opsworks][:stack][:id]
```

#### name

The stack name (string).

```
node[:opsworks][:stack][:name]
```

### rds\_instances

Contains a list of embedded objects, one for each Amazon RDS instance that is registered with the stack. Each embedded object contains a set of attributes that define the instance's configuration. You specify these values when you use the Amazon RDS console or API to create the instance. You can also use the Amazon RDS console or API to edit some of the settings after the instance has been created. For more information, see the [Amazon RDS documentation](#).

#### Note

The general node syntax for these attributes is as follows, where *i* specifies the instance's zero-based list index.

```
node[:opsworks][:stack][:rds_instances][i][:attribute_name]
```

If your stack has multiple Amazon RDS instances, the following is an example of how to use a particular instance in a recipe.

```
if my_rds = node[:opsworks][:stack][:rds_instances].select{|rds_instance|
  rds_instance[:db_instance_identifier] == 'db_id' }.first
  template "/etc/rds.conf" do
    source "rds.conf.erb"
    variables :address => my_rds[:address]
  end
end
```

<a href="#">address</a> (p. 387)	<a href="#">allocated_storage</a> (p. 387)	<a href="#">arn</a> (p. 388)
<a href="#">auto_minor_version_upgrade</a> (p.388)	<a href="#">availability_zone</a> (p. 388)	<a href="#">backup_retention_period</a> (p.388)
<a href="#">db_instance_class</a> (p. 388)	<a href="#">db_instance_identifier</a> (p. 388)	<a href="#">db_instance_status</a> (p. 388)
<a href="#">db_name</a> (p. 388)	<a href="#">db_parameter_groups</a> (p.388)	<a href="#">db_security_groups</a> (p. 389)
<a href="#">db_user</a> (p. 389)	<a href="#">engine</a> (p. 389)	<a href="#">instance_create_time</a> (p. 389)
<a href="#">license_model</a> (p. 389)	<a href="#">multi_az</a> (p. 389)	<a href="#">option_group_memberships</a> (p.390)
<a href="#">port</a> (p. 390)	<a href="#">preferred_backup_window</a> (p.390)	<a href="#">preferred_maintenance_window</a> (p.390)
<a href="#">publicly_accessible</a> (p. 390)	<a href="#">read_replica_db_instance_identifiers</a> (p.390)	<a href="#">region</a> (p. 390)
<a href="#">status_infos</a> (p. 390)	<a href="#">vpc_security_groups</a> (p. 391)	

### address

The instances URL, such as `opsinstance.ccdvt3hwogla.us-east-1.rds.amazonaws.com` (string).

```
node[:opsworks][:stack][:rds_instances][i][:address]
```

### allocated\_storage

The allocated storage, in GB (number).

```
node[:opsworks][:stack][:rds_instances][i][:allocated_storage]
```

**arn**

The instance's ARN (string).

```
node[:opsworks][:stack][:rds_instances][i][:arn]
```

**auto\_minor\_version\_upgrade**

Whether to automatically apply minor version upgrades (Boolean).

```
node[:opsworks][:stack][:rds_instances][i][:auto_minor_version_upgrade]
```

**availability\_zone**

The instance's Availability Zone, such as us-east-1a (string).

```
node[:opsworks][:stack][:rds_instances][i][:availability_zone]
```

**backup\_retention\_period**

The backup retention period, in days (number).

```
node[:opsworks][:stack][:rds_instances][i][:backup_retention_period]
```

**db\_instance\_class**

The DB instance class, such as db.m1.small (string).

```
node[:opsworks][:stack][:rds_instances][i][:db_instance_class]
```

**db\_instance\_identifier**

The user-defined DB instance identifier (string).

```
node[:opsworks][:stack][:rds_instances][i][:db_instance_identifier]
```

**db\_instance\_status**

The instance's status (string). For more information, see [DB Instance](#).

```
node[:opsworks][:stack][:rds_instances][i][:db_instance_status]
```

**db\_name**

The user-defined DB name (string).

```
node[:opsworks][:stack][:rds_instances][i][:db_name]
```

**db\_parameter\_groups**

The instance's DB parameter groups, which contains a list of embedded objects, one for each parameter group. For more information, see [Working with DB Parameter Groups](#). Each object contains the following attributes:

**db\_parameter\_group\_name**

The group name (string).

```
node[:opsworks][:stack][:rds_instances][i][:db_parameter_groups][j][:db_parameter_group_name]
```

**parameter\_apply\_status**

The apply status (string).

```
node[:opsworks][:stack][:rds_instances][i][:db_parameter_groups][j][:parameter_apply_status]
```

**db\_security\_groups**

The instance's database security groups, which contains a list of embedded objects, one for each security group. For more information, see [Working with DB Security Groups](#). Each object contains the following attributes

**db\_security\_group\_name**

The security group name (string).

```
node[:opsworks][:stack][:rds_instances][i][:db_security_groups][j][:db_security_group_name]
```

**status**

The status (string).

```
node[:opsworks][:stack][:rds_instances][i][:db_security_groups][j][:status]
```

**db\_user**

The user-defined Master User name (string).

```
node[:opsworks][:stack][:rds_instances][i][:db_user]
```

**engine**

The database engine, such as mysql(5.6.13) (string).

```
node[:opsworks][:stack][:rds_instances][i][:engine]
```

**instance\_create\_time**

The instance creation time, such as 2014-04-15T16:13:34Z (string).

```
node[:opsworks][:stack][:rds_instances][i][:instance_create_time]
```

**license\_model**

The instance's license model, such as general-public-license (string).

```
node[:opsworks][:stack][:rds_instances][i][:license_model]
```

**multi\_az**

Whether multi-AZ deployment is enabled (Boolean).

```
node[:opsworks][:stack][:rds_instances][i][:multi_az]
```

**option\_group\_memberships**

The instance's option group memberships, which contains a list of embedded objects, one for each option group. For more information, see [Working with Option Groups](#). Each object contains the following attributes:

**option\_group\_name**

The group's name (string).

```
node[:opsworks][:stack][:rds_instances][i][:option_group_memberships][j][:option_group_name]
```

**status**

The group's status (string).

```
node[:opsworks][:stack][:rds_instances][i][:option_group_memberships][j][:status]
```

**port**

The database server's port (number).

```
node[:opsworks][:stack][:rds_instances][i][:port]
```

**preferred\_backup\_window**

The preferred daily backup window, such as 06:26-06:56 (string).

```
node[:opsworks][:stack][:rds_instances][i][:preferred_backup_window]
```

**preferred\_maintenance\_window**

The preferred weekly maintenance window, such as thu:07:13-thu:07:43 (string).

```
node[:opsworks][:stack][:rds_instances][i][:preferred_maintenance_window]
```

**publicly\_accessible**

Whether the database is publicly accessible (Boolean).

```
node[:opsworks][:stack][:rds_instances][i][:publicly_accessible]
```

**read\_replica\_db\_instance\_identifiers**

A list of the read-replica instance identifiers (list of string). For more information, see [Working with Read Replicas](#).

```
node[:opsworks][:stack][:rds_instances][i][:read_replica_db_instance_identifiers]
```

**region**

The AWS region, such as us-east-1 (string).

```
node[:opsworks][:stack][:rds_instances][i][:region]
```

**status\_infos**

A list of status information (list of string).

```
node[:opsworks][:stack][:rds_instances][i][:status_infos]
```

**vpc\_security\_groups**

A list of VPC security groups (list of string).

```
node[:opsworks][:stack][:rds_instances][i][:vpc_security_groups]
```

**vpc\_id**

The VPC id (string). This value is `null` if the instance is not in a VPC.

```
node[:opsworks][:stack][:vpc_id]
```

## Other Top-level opsworks Attributes

This section contains the `opsworks` attributes that do not have child attributes.

**activity**

The activity that is associated with the JSON object (string).

```
node[:opsworks][:activity]
```

**agent\_version**

The version of the instance's OpsWorks agent (string).

```
node[:opsworks][:agent_version]
```

**deploy\_chef\_provider**

The Chef deploy provider, which influences a deployed app's directory structure (string). For more information, see [deploy](#). You can set this attribute to one the following:

- Branch
- Revision
- Timestamped (default value)

```
node[:opsworks][:deploy_chef_provider]
```

**ruby\_stack**

The Ruby stack (string). The default setting is the enterprise version (`ruby_enterprise`). For the MRI version, set this attribute to `ruby`.

```
node[:opsworks][:ruby_stack]
```

**ruby\_version**

The Ruby version number (string).

```
node[:opsworks][:ruby_version]
```

**run\_cookbook\_tests**

Whether to run [minitest-chef-handler](#) tests on your Chef 11.4 cookbooks (Boolean).

```
node[ :opsworks ] [ :run_cookbook_tests ]
```

**sent\_at**

When this command was sent to the instance (number).

```
node[ :opsworks ] [ :sent_at ]
```

**deployment**

If this JSON is associated with a deploy activity, deployment is set to the deployment ID, an AWS OpsWorks-generated GUID that uniquely identifies the deployment (string). Otherwise the attribute is set to null.

```
node[ :opsworks ] [ :deployment ]
```

## opsworks\_custom\_cookbooks Attributes

Contains attributes that specify the stack's custom cookbooks.

**enabled**

Whether custom cookbooks are enabled (Boolean).

```
node[ :opsworks_custom_cookbooks ] [ :enabled ]
```

**recipes**

A list of the recipes that are to be executed for this command, including custom recipes, using the `cookbookname` `::recipename` format (list of string).

```
node[ :opsworks_custom_cookbooks ] [ :recipes ]
```

## dependencies Attributes

Contains several attributes that are related to the [update\\_dependencies stack command \(p. 52\)](#).

**gem\_binary**

The location of the Gems binary (string).

**upgrade\_debs**

Whether to upgrade Debs packages (Boolean).

**update\_debs**

Whether to update Debs packages (Boolean).

## ganglia Attributes

Contains a **web** attribute that contains several attributes that specify how to access the Ganglia statistics web page:

**password**

The password required to access the statistics page (string).

```
node[:ganglia][:web][:password]
```

**url**

The statistics page's URL path, such as "/ganglia" (string). The complete URL is `http://DNS-NameURLPath`, where `DNSName` is the associated instance's DNS name.

```
node[:ganglia][:web][:url]
```

**user**

The user name required to access the statistics page (string).

```
node[:ganglia][:web][:user]
```

## mysql Attributes

Contains a set of attributes that specify the MySQL database server configuration.

**clients**

A list of client IP addresses (list of string).

```
node[:mysql][:clients]
```

**server\_root\_password**

The root password (string).

```
node[:mysql][:server_root_password]
```

## passenger Attributes

Contains a set of attributes that specify the Phusion Passenger configuration.

**gem\_bin**

The location of the RubyGems binaries, such as "/usr/local/bin/gem" (string).

```
node[:passenger][:gem_bin]
```

**max\_pool\_size**

The maximum pool size (number).

```
node[:passenger][:max_pool_size]
```

**ruby\_bin**

The location of the Ruby binaries, such as "/usr/local/bin/ruby".

```
node[:passenger][:ruby_bin]
```

**version**

The Passenger version (string).

```
node[ :passenger ][ :version ]
```

## opsworks\_bundler Attributes

Contains elements that specify [Bundler](#) support.

### manage\_package

Whether to install and manage Bundler (Boolean).

```
node[ :opsworks_bundler ][ :manage_package ]
```

### version

The bundler version (string).

```
node[ :opsworks_bundler ][ :version ]
```

## deploy Attributes

If the JSON is associated with a [Deploy event \(p. 176\)](#) or an [Execute Recipes stack command \(p. 52\)](#), the `deploy` attribute contains an embedded JSON object for each app that was deployed, named by the app's short name. Each object contains the following attributes:

<a href="#">application (p. 394)</a>	<a href="#">application_type (p. 394)</a>	<a href="#">auto_bundle_on_deploy (p. 395)</a>
<a href="#">database (p. 395)</a>	<a href="#">deploy_to (p. 395)</a>	<a href="#">domains (p. 395)</a>
<a href="#">document_root (p. 396)</a>	<a href="#">environment_variables (p. 396)</a>	<a href="#">group (p. 396)</a>
<a href="#">keep_releases (p. 396)</a>	<a href="#">memcached (p. 396)</a>	<a href="#">migrate (p. 396)</a>
<a href="#">mounted_at (p. 396)</a>	<a href="#">rails_env (p. 397)</a>	<a href="#">restart_command (p. 397)</a>
<a href="#">scm (p. 397)</a>	<a href="#">ssl_certificate (p. 398)</a>	<a href="#">ssl_certificate_ca (p. 398)</a>
<a href="#">ssl_certificate_key (p. 398)</a>	<a href="#">ssl_support (p. 398)</a>	<a href="#">stack (p. 398)</a>
<a href="#">symlink_before_migrate (p. 398)</a>	<a href="#">symlinks (p. 398)</a>	<a href="#">user (p. 398)</a>

### application

The app's slug name, such as "simplephp" (string).

```
node[ :deploy ][ 'appshortname' ][ :application ]
```

### application\_type

The app type (string). Possible values are as follows:

- `java`: A Java app
- `nodejs`: A Node.js app
- `php`: A PHP app
- `rails`: A Ruby on Rails app
- `web`: A static HTML page

- other: All other application types

```
node[:deploy]['appshortname'][:application_type]
```

**auto\_bundle\_on\_deploy**

For Rails applications, whether to execute bundler during the deployment (Boolean).

```
node[:deploy]['appshortname'][:auto_bundle_on_deploy]
```

**database**

If the stack includes a [database layer \(p. 77\)](#), contains the information required to connect to database adapter

The database adapter, such as `mysql` (string).

**database**

The database name, which is usually the app's slug name, such as "simplephp" (string).

```
node[:deploy]['appshortname'][:database][:database]
```

**data\_source\_provider**

The data source: `mysql` or `rds` (string).

**host**

The database host's IP address (string).

```
node[:deploy]['appshortname'][:database][:host]
```

**password**

The database password (string).

```
node[:deploy]['appshortname'][:database][:password]
```

**port**

The database port (number).

**reconnect**

For Rails applications, whether the application should reconnect if the connection no longer exists (Boolean).

```
node[:deploy]['appshortname'][:database][:reconnect]
```

**username**

The user name (string).

```
node[:deploy]['appshortname'][:database][:username]
```

**deploy\_to**

Where the app is to be deployed to, such as "/srv/www/simplephp" (string).

```
node[:deploy]['appshortname'][:deploy_to]
```

**domains**

A list of the app's domains (list of string).

```
node[:deploy]['appshortname'][:domains]
```

**document\_root**

The document root, if you specify a nondefault root, or null if you use the default root (string).

```
node[:deploy]['appshortname'][:document_root]
```

**environment\_variables**

A collection of up to ten attributes that represent the user-specified environment variables that have been defined for the app. For more information about how to define an app's environment variables, see [Adding Apps \(p. 125\)](#). Each attribute name is set to an environment variable name and the corresponding value is set to the variable's value, so you can use the following syntax to reference a particular value.

```
node[:deploy]['appshortname'][:environment_variables][:variable_name]
```

**group**

The app's group (string).

```
node[:deploy]['appshortname'][:group]
```

**keep\_releases**

The number of app deployments that AWS OpsWorks will store (number). This attribute controls the number of times you can roll back an app. By default, it is set to the global value, [deploy\\_keep\\_releases \(p. 406\)](#), which has a default value of 5. You can override `keep_releases` to specify the number of stored deployments for a particular application.

```
node[:deploy]['appshortname'][:keep_releases]
```

**memcached**

Contains two attributes that define the memcached configuration.

**host**

The Memcached server instance's IP address (string).

```
node[:deploy]['appshortname'][:memcached][:host]
```

**port**

The port that the memcached server is listening on (number).

```
node[:deploy]['appshortname'][:memcached][:port]
```

**migrate**

For Rails applications, whether to run migrations (Boolean).

```
node[:deploy]['appshortname'][:migrate]
```

**mounted\_at**

The app's mount point, if you specify a nondefault mount point, or null if you use the default mount point (string).

```
node[:deploy]['appshortname'][:mounted_at]
```

**rails\_env**

For Rails App Server instances, the rails environment, such as "production" (string).

```
node[:deploy]['appshortname'][:rails_env]
```

**restart\_command**

A command to be run when the app is restarted, such as "echo 'restarting app'".

```
node[:deploy]['appshortname'][:restart_command]
```

**scm**

Contains a set of attributes that specify the information that OpsWorks uses to deploy the app from its source control repository. The attributes vary depending on the repository type.

**password**

The password, for private repositories, and null for public repositories (string). For private Amazon S3 buckets, the attribute is set to the secret key.

```
node[:deploy]['appshortname'][:scm][:password]
```

**repository**

The repository URL, such as "git://github.com/amazonwebservices/opsworks-demo-php-simple-app.git" (string).

```
node[:deploy]['appshortname'][:scm][:repository]
```

**revision**

If the repository has multiple branches, the attribute specifies the app's branch or version, such as "version1" (string). Otherwise it is set to null.

```
node[:deploy]['appshortname'][:scm][:revision]
```

**scm\_type**

The repository type (string). Possible values are as follows:

- "git": A Git repository
- "svn": A Subversion repository
- "s3": An Amazon S3 bucket
- "archive": An HTTP archive
- "other": Another repository type

```
node[:deploy]['appshortname'][:scm][:scm_type]
```

**ssh\_key**

A [deploy SSH key \(p. 146\)](#), for accessing private Git repositories, and null for public repositories (string).

```
node[:deploy]['appshortname'][:scm][:ssh_key]
```

**user**

The user name, for private repositories, and null for public repositories (string). For private Amazon S3 buckets, the attribute is set to the access key.

```
node[:deploy]['appshortname'][:scm][:user]
```

**ssl\_certificate**

The app's SSL certificate, if you enabled SSL support, or null otherwise (string).

```
node[:deploy]['appshortname'][:ssl_certificate]
```

**ssl\_certificate\_ca**

If SSL is enabled, an attribute for specifying an intermediate certificate authority key or client authentication (string).

```
node[:deploy]['appshortname'][:ssl_certificate_ca]
```

**ssl\_certificate\_key**

The app's SSL private key, if you enabled SSL support, or null otherwise (string).

```
node[:deploy]['appshortname'][:ssl_certificate_key]
```

**ssl\_support**

Whether SSL is supported (Boolean).

```
node[:deploy]['appshortname'][:ssl_support]
```

**stack**

Contains one Boolean attribute, needs\_reload, that specifies whether to reload the app server during deployment.

```
node[:deploy]['appshortname'][:stack][:needs_reload]
```

**symlink\_before\_migrate**

For Rails apps, contains symlinks that are to be created before running migrations as "link" : "target" pairs.

```
node[:deploy]['appshortname'][:symlink_before_migrate]
```

**symlinks**

Contains the deployment's symlinks as "link" : "target" pairs.

```
node[:deploy]['appshortname'][:symlinks]
```

**user**

The app's user (string).

```
node[:deploy]['appshortname'][:user]
```

## Other Top-Level Attributes

This section contains top-level stack configuration attributes that do not have child attributes.

### **rails Attributes**

Contains a **max\_pool\_size** attribute that specifies the server's maximum pool size (number).

```
node[ :rails][ :max_pool_size]
```

### **recipes Attributes**

A list of the built-in recipes that were run by this activity, using the "*cookbookname*::*recipename*" format (list of string).

```
node[ :recipes]
```

### **opsworks\_rubygems Attributes**

Contains a **version** element that specifies the RubyGems version (string).

```
node[ :opsworks_rubygems][ :version]
```

### **languages Attributes**

Contains an attribute for each installed language, named for the language, such as **ruby**. The attribute is an object that contains an attribute, such as **ruby\_bin**, that specifies the installation folder, such as "/usr/bin/ruby" (string).

### **ssh\_users Attributes**

Contains a set of attributes, each of which describes one of the IAM users that have been granted SSH permissions. Each attribute is named with a user's Unix ID. AWS OpsWorks generates a unique ID for each user in the 2000 range, such as "2001", and creates a user with that ID on every instance. Each attribute contains the following attributes:

#### **email**

The IAM user's e-mail address (string).

```
node[ :ssh_users][ 'UnixID' ][ :email]
```

#### **public\_key**

The IAM user's public SSH key (string).

```
node[ :ssh_users][ 'UnixID' ][ :public_key]
```

#### **sudoer**

Whether the IAM user has sudo permissions (Boolean).

```
node[ :ssh_users][ 'UnixID' ][ :sudoer]
```

#### **name**

The IAM user name (string).

```
node[ :ssh_users][ 'UnixID' ][ :name]
```

# Built-in Recipe Attributes

Most of the built-in recipes have one or more [attributes files \(p. 156\)](#) that define various settings. You can access these settings in your custom recipes and use custom JSON to override them. You typically need to access or override attributes that control the configuration of the various server technologies that are supported by AWS OpsWorks. This section summarizes those attributes. The complete attributes files, and the associated recipes and templates, are available at <https://github.com/aws/opsworks-cookbooks.git>.

## Note

All built-in recipe attributes are `default` type.

## Topics

- [apache2 Attributes \(p. 400\)](#)
- [deploy Attributes \(p. 406\)](#)
- [haproxy Attributes \(p. 407\)](#)
- [memached Attributes \(p. 410\)](#)
- [mysql Attributes \(p. 411\)](#)
- [nginx Attributes \(p. 415\)](#)
- [opsworks\\_java Attributes \(p. 418\)](#)
- [passenger\\_apache2 Attributes \(p. 421\)](#)
- [ruby Attributes \(p. 423\)](#)
- [unicorn Attributes \(p. 424\)](#)

## apache2 Attributes

The `apache2` attributes specify the Apache HTTP server configuration. For more information, see [Apache Core Features](#).

<a href="#">binary (p. 400)</a>	<a href="#">contact (p. 400)</a>	<a href="#">deflate_types (p. 401)</a>
<a href="#">dir (p. 401)</a>	<a href="#">document_root (p. 401)</a>	<a href="#">group (p. 401)</a>
<a href="#">hide_info_headers (p. 402)</a>	<a href="#">icondir (p. 402)</a>	<a href="#">init_script (p. 402)</a>
<a href="#">keepalive (p. 402)</a>	<a href="#">keepaliverequests (p. 402)</a>	<a href="#">keepalivetimeout (p. 402)</a>
<a href="#">lib_dir (p. 402)</a>	<a href="#">libexecdir (p. 402)</a>	<a href="#">listen_ports (p. 403)</a>
<a href="#">log_dir (p. 403)</a>	<a href="#">logrotate Attributes (p. 403)</a>	<a href="#">pid_file (p. 404)</a>
<a href="#">prefork Attributes (p. 404)</a>	<a href="#">serversignature (p. 404)</a>	<a href="#">servertokens (p. 405)</a>
<a href="#">timeout (p. 405)</a>	<a href="#">traceenable (p. 405)</a>	<a href="#">user (p. 405)</a>
<a href="#">worker Attributes (p. 405)</a>		

### binary

The location of the Apache binary (string). The default value is '`/usr/sbin/httpd`'.

```
node[ :apache] [ :binary]
```

### contact

An e-mail contact (string). The default value is a dummy address, '`ops@example.com`'.

```
node[:apache][:contact]
```

#### **deflate\_types**

Directs mod\_deflate to enable compression for the specified Mime types, if they are supported by the browser (list of string). The default value is as follows:

```
['application/javascript',
 'application/json',
 'application/x-javascript',
 'application/xhtml+xml',
 'application/xml',
 'application/xml+rss',
 'text/css',
 'text/html',
 'text/javascript',
 'text/plain',
 'text/xml']
```

#### **Caution**

Compression can introduce security risks. To completely disable compression, set this attribute as follows:

```
node[:apache][:deflate_types] = []
```

```
node[:apache][:deflate_types]
```

#### **dir**

The server's root directory (string). The default values are as follows:

- Amazon Linux: '/etc/httpd'
- Ubuntu: '/etc/apache2'

```
node[:apache][:dir]
```

#### **document\_root**

The document root (string). The default values are as follows:

- Amazon Linux: '/var/www/html'
- Ubuntu: '/var/www'

```
node[:apache][:document_root]
```

#### **group**

The group name (string). The default values are as follows:

- Amazon Linux: 'apache'
- Ubuntu: 'www-data'

```
node[:apache][:group]
```

#### hide\_info\_headers

Whether to omit version and module information from HTTP headers ('true'/'false') (string).  
The default value is 'true'.

```
node[:apache][:hide_info_headers]
```

#### icondir

The icon directory (string). The defaults value are as follows:

- Amazon Linux: '/var/www/icons/'
- Ubuntu: '/usr/share/apache2/icons'

```
node[:apache][:icondir]
```

#### init\_script

The initialization script (string). The default values are as follows:

- Amazon Linux: '/etc/init.d/httpd'
- Ubuntu: '/etc/init.d/apache2'

```
node[:apache][:init_script]
```

#### keepalive

Whether to enable keep-alive connections (string). The possible values are 'On' and 'Off' (string).  
The default value is 'Off'.

```
node[:apache][:keepalive]
```

#### keepaliverequests

The maximum number of keep-alive requests that Apache will handle at the same time (number).  
The default value is 100.

```
node[:apache][:keepaliverequests]
```

#### keepalivetimeout

The time that Apache waits for a request before closing the connection (number). The default value is 3.

```
node[:apache][:keepalivetimeout]
```

#### lib\_dir

The directory that contains the object code libraries (string). The default values are as follows:

- Amazon Linux (x86): '/usr/lib/httpd'
- Amazon Linux (x64): '/usr/lib64/httpd'
- Ubuntu: '/usr/lib/apache2'

```
node[:apache][:lib_dir]
```

#### libexecdir

The directory that contains the program executables (string). The default values are as follows:

- Amazon Linux (x86): '/usr/lib/httpd/modules'
- Amazon Linux (x64): '/usr/lib64/httpd/modules'

- Ubuntu: '/usr/lib/apache2/modules'

```
node[:apache][:libexecdir]
```

#### listen\_ports

A list of ports that the server listens to (list of string). The default value is [ '80', '443' ].

```
node[:apache][:listen_ports]
```

#### log\_dir

The log directory (string). The default values are as follows:

- Amazon: '/var/log/httpd'
- Ubuntu: '/var/log/apache2'

```
node[:apache][:log_dir]
```

### logrotate Attributes

These attributes specify how to rotate the log files.

#### delaycompress

Whether to delay compressing a closed log file until the start of the next rotation cycle ('true'/'false') (string). The default value is 'true'.

```
node[:apache][:logrotate][:delaycompress]
```

#### group

The log files' group (string). The default value is 'adm'.

```
node[:apache][:logrotate][:group]
```

#### mode

The log files' mode (string). The default value is '640'.

```
node[:apache][:logrotate][:mode]
```

#### owner

The log files' owner (string). The default value is 'root'.

```
node[:apache][:logrotate][:owner]
```

#### rotate

The number of rotation cycles before a closed log file is removed (string). The default value is '30'.

```
node[:apache][:logrotate][:rotate]
```

#### schedule

The rotation schedule (string). Possible values are as follows:

- 'daily'
- 'weekly'
- 'monthly'

The default value is 'daily'.

```
node[ :apache][:logrotate][:schedule]
```

#### pid\_file

The file that contains the daemon's process ID (string). The default values are as follows:

- Amazon (Linux version 6 or later): '/var/run/httpd/httpd.pid'
- Amazon (Linux version 5 or earlier): '/var/run/httpd.pid'
- Ubuntu: '/var/run/apache2.pid'

```
node[ :apache][:pid_file]
```

#### prefork Attributes

These attributes specify the pre-forking configuration.

##### maxclients

The maximum number of simultaneous requests that will be served (number). The default value is 400.

```
node[ :apache][:prefork][:maxclients]
```

##### maxrequestsperchild

The maximum number of requests that a child server process will handle (number). The default value is 10000.

```
node[ :apache][:prefork][:maxrequestsperchild]
```

##### maxspareservers

The maximum number of idle child server processes (number). The default value is 32.

```
node[ :apache][:prefork][:maxspareservers]
```

##### minspareservers

The minimum number of idle child server processes (number). The default value is 16.

```
node[ :apache][:prefork][:minspareservers]
```

##### serverlimit

The maximum number of processes that can be configured (number). The default value is 400.

```
node[ :apache][:prefork][:serverlimit]
```

##### startservers

The number of child server processes to be created at startup (number). The default value is 16.

```
node[ :apache][:prefork][:startservers]
```

##### serversignature

Specifies whether and how to configure a trailing footer for server-generated documents (string). The possible values are 'On', 'Off', and 'Email'). The default value is 'Off'.

```
node[:apache][:serversignature]
```

#### **servertokens**

Specifies what type of server version information is included in the response header (string):

- 'Full': Full information. For example, Server: Apache/2.4.2 (Unix) PHP/4.2.2 MyMod/1.2
- 'Prod': Product name. For example, Server: Apache
- 'Major': Major version. For example, Server: Apache/2
- 'Minor': Major and minor version. For example, Server: Apache/2.4
- 'Min': Minimal version. For example, Server: Apache/2.4.2
- 'OS': Version with operating system. For example, Server: Apache/2.4.2 (Unix)

The default value is 'Prod'.

```
node[:apache][:servertokens]
```

#### **timeout**

The amount of time that Apache waits for I/O (number). The default value is 120.

```
node[:apache][:timeout]
```

#### **traceenable**

Whether to enable TRACE requests (string). The possible values are 'On' and 'Off'. The default value is 'Off'.

```
node[:apache][:traceenable]
```

#### **user**

The user name (string). The default values are as follows:

- Amazon Linux: 'apache'
- Ubuntu: 'www-data'

```
node[:apache][:user]
```

### **worker Attributes**

These attributes specify the worker process configuration.

#### **startservers**

The number of child server processes to be created at startup (number). The default value is 4.

```
node[:apache][:worker][:startservers]
```

#### **maxclients**

The maximum number of simultaneous requests that will be served (number). The default value is 1024.

```
node[:apache][:worker][:maxclients]
```

#### **maxsparethreads**

The maximum number of idle threads (number). The default value is 192.

```
node[ :apache ][ :worker ][ :maxsparethreads ]
```

**minsparethreads**

The minimum number of idle threads (number). The default value is 64.

```
node[ :apache ][ :worker ][ :minsparethreads ]
```

**threadsperchild**

The number of threads per child process (number). The default value is 64.

```
node[ :apache ][ :worker ][ :threadsperchild ]
```

**maxrequestsperchild**

The maximum number of requests that a child server process will handle (number). The default value is 10000.

```
node[ :apache ][ :worker ][ :maxrequestsperchild ]
```

## deploy Attributes

The [built-in deploy cookbook's deploy.rb attributes file](#) defines the following attributes in the `opsworks` namespace. For more information on deploy directories, see [Deploy Recipes \(p. 254\)](#).

**deploy\_keep\_releases**

A global setting for the number of app deployments that AWS OpsWorks will store (number). The default value is 5. This value controls the number of times you can roll back an app.

```
node[ :opsworks ][ :deploy_keep_releases ]
```

**group**

The `group` setting for the app's deploy directory (string). The default value depends on the instance's operating system:

- For Ubuntu instances, the default value is `www-data`.
- For Amazon Linux instance that are members of a Rails App Server layer that uses Nginx and Unicorn, the default value is `nginx`.
- For all other Amazon Linux instances, the default value is `apache`.

```
node[ :opsworks ][ :deploy_user ][ :group ]
```

**user**

The `user` setting for the app's deploy directory (string). The default value is `deploy`.

```
node[ :opsworks ][ :deploy_user ][ :user ]
```

## haproxy Attributes

The [haproxy attributes](#) specify the HAProxy server configuration. For more information, see [HAProxy Docs](#).

<a href="#">balance</a> (p. 407)	<a href="#">check_interval</a> (p. 407)	<a href="#">client_timeout</a> (p. 407)
<a href="#">connect_timeout</a> (p. 408)	<a href="#">default_max_connections</a> (p. 408)	<a href="#">global_max_connections</a> (p. 408)
<a href="#">health_check_method</a> (p. 408)	<a href="#">health_check_url</a> (p. 408)	<a href="#">queue_timeout</a> (p. 408)
<a href="#">http_request_timeout</a> (p. 408)	<a href="#">maxcon_factor_nodejs_app</a> (p. 409)	<a href="#">maxcon_factor_nodejs_app_ssl</a> (p. 409)
<a href="#">maxcon_factor_php_app</a> (p. 409)	<a href="#">maxcon_factor_php_app_ssl</a> (p. 409)	<a href="#">maxcon_factor_rails_app</a> (p. 409)
<a href="#">maxcon_factor_rails_app_ssl</a> (p. 409)	<a href="#">maxcon_factor_static</a> (p. 409)	<a href="#">maxcon_factor_static_ssl</a> (p. 410)
<a href="#">retries</a> (p. 408)	<a href="#">server_timeout</a> (p. 408)	<a href="#">stats_url</a> (p. 408)
<a href="#">stats_user</a> (p. 409)		

### balance

The algorithm used by a load balancer to select a server (string). The default value is 'roundrobin'. The other options are:

- 'static-rr'
- 'leastconn'
- 'source'
- 'uri'
- 'url\_param'
- 'hdr(name)'
- 'rdp-cookie'
- 'rdp-cookie(name)'

For more information on these arguments, see [balance](#).

```
node[ :haproxy][ :balance]
```

### check\_interval

The health check time interval (string). The default value is '10s'.

```
node[ :haproxy][ :check_interval]
```

### client\_timeout

The maximum amount of time that a client can be inactive (string). The default value is '60s'.

```
node[ :haproxy][ :client_timeout]
```

**connect\_timeout**

The maximum amount of time that HAProxy will wait for a server connection attempt to succeed (string). The default value is '10s'.

```
node[ :haproxy ][ :connect_timeout ]
```

**default\_max\_connections**

The default maximum number of connections (string). The default value is '80000'.

```
node[ :haproxy ][ :default_max_connections ]
```

**global\_max\_connections**

The maximum number of connections (string). The default value is '80000'.

```
node[ :haproxy ][ :global_max_connections ]
```

**health\_check\_method**

The health check method (string). The default value is 'OPTIONS'.

```
node[ :haproxy ][ :health_check_method ]
```

**health\_check\_url**

The URL path that is used to check servers' health (string). The default value is '/'.

```
node[ :haproxy ][ :health_check_url ]
```

**queue\_timeout**

The maximum wait time for a free connection (string). The default value is '120s'.

```
node[ :haproxy ][ :queue_timeout ]
```

**http\_request\_timeout**

The maximum amount of time that HAProxy will wait for a complete HTTP request (string). The default value is '30s'.

```
node[ :haproxy ][ :http_request_timeout ]
```

**retries**

The number of retries after server connection failure (string). The default value is '3'.

```
node[ :haproxy ][ :retries ]
```

**server\_timeout**

The maximum amount of time that a client can be inactive (string). The default value is '60s'.

```
node[ :haproxy ][ :server_timeout ]
```

**stats\_url**

The URL path for the statistics page (string). The default value is '/haproxy?stats'.

```
node[:haproxy][:stats_url]
```

**stats\_user**

The statistics page user name (string). The default value is 'opsworks'.

```
node[:haproxy][:stats_user]
```

The `maxcon` attributes represent a load factor multiplier that is used to compute the maximum number of connections that HAProxy allows for [backends \(p. 381\)](#). For example, suppose you have a Rails app server on a small instance with a backend value of 4, which means that AWS OpsWorks will configure four Rails processes for that instance. If you use the default `maxcon_factor_rails_app` value of 7, HAProxy will handle 28 ( $4 \times 7$ ) connections to the Rails server.

**maxcon\_factor\_nodejs\_app**

The maxcon factor for a Node.js app server (number). The default value is 10.

```
node[:haproxy][:maxcon_factor_nodejs_app]
```

**maxcon\_factor\_nodejs\_app\_ssl**

The maxcon factor for a Node.js app server with SSL (number). The default value is 10.

```
node[:haproxy][:maxcon_factor_nodejs_app_ssl]
```

**maxcon\_factor\_php\_app**

The maxcon factor for a PHP app server (number). The default value is 10.

```
node[:haproxy][:maxcon_factor_php_app]
```

**maxcon\_factor\_php\_app\_ssl**

The maxcon factor for a PHP app server with SSL (number). The default value is 10.

```
node[:haproxy][:maxcon_factor_php_app_ssl]
```

**maxcon\_factor\_rails\_app**

The maxcon factor for a Rails app server (number). The default value is 7.

```
node[:haproxy][:maxcon_factor_rails_app]
```

**maxcon\_factor\_rails\_app\_ssl**

The maxcon factor for a Rails app server with SSL (number). The default value is 7.

```
node[:haproxy][:maxcon_factor_rails_app_ssl]
```

**maxcon\_factor\_static**

The maxcon factor for a static web server (number). The default value is 15.

```
node[:haproxy][:maxcon_factor_static]
```

**maxcon\_factor\_static\_ssl**

The maxcon factor for a static web server with SSL (number). The default value is 15.

```
node[ :haproxy][ :maxcon_factor_static_ssl]
```

## memached Attributes

The [memached attributes](#) specify the [Memcached](#) server configuration.

<a href="#">memory</a> (p. 410)	<a href="#">max_connections</a> (p. 410)	<a href="#">pid_file</a> (p. 410)
<a href="#">port</a> (p. 410)	<a href="#">start_command</a> (p. 410)	<a href="#">stop_command</a> (p. 410)
<a href="#">user</a> (p. 410)		

**memory**

The maximum memory to use, in MB (number). The default value is 512.

```
node[ :memcached][ :memory]
```

**max\_connections**

The maximum number of connections (string). The default value is '4096'.

```
node[ :memcached][ :max_connections]
```

**pid\_file**

The file that contains the daemon's process ID (string). The default value is 'var/run/memcached.pid'.

```
node[ :memcached][ :pid_file]
```

**port**

The port to listen on (number). The default value is 11211.

```
node[ :memcached][ :port]
```

**start\_command**

The start command (string). The default value is '/etc/init.d/memcached start'.

```
node[ :memcached][ :start_command]
```

**stop\_command**

The stop command (string). The default value is '/etc/init.d/memcached stop'.

```
node[ :memcached][ :stop_command]
```

**user**

The user (string). The default value is 'nobody'.

```
node[ :memcached][ :user]
```

## mysql Attributes

The [mysql attributes](#) specify the MySQL master configuration. For more information, see [Server System Variables](#).

<a href="#">basedir (p. 411)</a>	<a href="#">bind_address (p. 411)</a>	<a href="#">clients (p. 411)</a>
<a href="#">conf_dir (p. 411)</a>	<a href="#">confd_dir (p. 411)</a>	<a href="#">datadir (p. 411)</a>
<a href="#">grants_path (p. 412)</a>	<a href="#">mysql_bin (p. 412)</a>	<a href="#">mysqladmin_bin (p. 412)</a>
<a href="#">pid_file (p. 412)</a>	<a href="#">port (p. 412)</a>	<a href="#">root_group (p. 412)</a>
<a href="#">server_root_password (p. 412)</a>	<a href="#">socket (p. 412)</a>	<a href="#">tunable Attributes (p. 412)</a>

### basedir

The base directory (string). The default value is '/usr'.

```
node[ :mysql][ :basedir]
```

### bind\_address

The address that MySQL listens on (string). The default value is '0.0.0.0'.

```
node[ :mysql][ :bind_address]
```

### clients

A list of clients (list of string).

```
node[ :mysql][ :clients]
```

### conf\_dir

The directory that contains the configuration file (string). The default values are as follows:

- Amazon: '/etc'
- Ubuntu: '/etc/mysql'

```
node[ :mysql][ :conf_dir]
```

### confd\_dir

The directory that contains additional configuration files (string). The default value is '/etc/mysql/conf.d'.

```
node[ :mysql][ :confd_dir]
```

### datadir

The data directory (string). The default value is '/var/lib/mysql'.

```
node[ :mysql][ :datadir]
```

**grants\_path**

The grant table location (string). The default value is '/etc/mysql\_grants.sql'.

```
node[ :mysql][ :grants_path]
```

**mysql\_bin**

The mysql binaries location (string). The default value is '/usr/bin/mysql'.

```
node[ :mysql][ :mysql_bin]
```

**mysqladmin\_bin**

The mysqladmin location (string). The default value is '/usr/bin/mysqladmin'.

```
node[ :mysql][ :mysqladmin_bin]
```

**pid\_file**

The file that contains the daemon's process ID (string). The default value is '/var/run/mysqld/mysqld.pid'.

```
node[ :mysql][ :pid_file]
```

**port**

The port that the server listens on (number). The default value is 3306.

```
node[ :mysql][ :port]
```

**root\_group**

The root group (string). The default value is 'root'.

```
node[ :mysql][ :root_group]
```

**server\_root\_password**

The server's root password (string). The default value is randomly generated.

```
node[ :mysql][ :server_root_password]
```

**socket**

The location of the socket file (string). The default value is '/var/lib/mysql/mysql.sock'. The default values are as follows:

- Amazon Linux: '/var/lib/mysql/mysql.sock'
- Ubuntu: '/var/run/mysqld/mysqld.sock'

```
node[ :mysql][ :socket]
```

**tunable Attributes**

The tunable attributes are used for performance tuning.

[back\\_log](#) (p. 413)

[innodb\\_additional\\_mem\\_pool\\_size](#)  
(p. 413)

[innodb\\_buffer\\_pool\\_size](#)  
(p. 413)

<a href="#">innodb_flush_log_at_trx_commit</a> (p. 413)	<a href="#">innodb_lock_wait_timeout</a> (p. 413)	<a href="#">key_buffer</a> (p. 413)
<a href="#">log_slow_queries</a> (p. 413)	<a href="#">long_query_time</a> (p. 414)	<a href="#">max_allowed_packet</a> (p. 414)
<a href="#">max_connections</a> (p. 414)	<a href="#">max_heap_table_size</a> (p. 414)	<a href="#">net_read_timeout</a> (p. 414)
<a href="#">net_write_timeout</a> (p. 414)	<a href="#">query_cache_limit</a> (p. 414)	<a href="#">query_cache_size</a> (p. 414)
<a href="#">query_cache_type</a> (p. 414)	<a href="#">thread_cache_size</a> (p. 415)	<a href="#">thread_stack</a> (p. 415)
<a href="#">wait_timeout</a> (p. 415)	<a href="#">table_cache</a> (p. 415)	

### **back\_log**

The maximum number of outstanding requests (string). The default value is '128'.

```
node[:mysql][:tunable][:back_log]
```

### **innodb\_additional\_mem\_pool\_size**

The size of the pool that [Innodb](#) uses to store internal data structures (string). The default value is '20M'.

```
node[:mysql][:tunable][:innodb_additional_mem_pool_size]
```

### **innodb\_buffer\_pool\_size**

The [Innodb](#) buffer pool size (string). The default value is '1200M'.

```
node[:mysql][:tunable][:innodb_buffer_pool_size]
```

### **innodb\_flush\_log\_at\_trx\_commit**

How often [Innodb](#) flushes the log buffer (string). The default value is '2'. For more information, see [innodb\\_flush\\_log\\_at\\_trx\\_commit](#).

```
node[:mysql][:tunable][:innodb_flush_log_at_trx_commit]
```

### **innodb\_lock\_wait\_timeout**

The maximum amount of time, in seconds, that an [Innodb](#) transaction waits for a row lock (string). The default value is '50'.

```
node[:mysql][:tunable][:innodb_lock_wait_timeout]
```

### **key\_buffer**

The index buffer size (string). The default value is '250M'.

```
node[:mysql][:tunable][:key_buffer]
```

### **log\_slow\_queries**

The location of the slow-query log file (string). The default value is '/var/log/mysql/mysql-slow.log'.

```
node[:mysql][:tunable][:log_slow_queries]
```

**long\_query\_time**

The time, in seconds, required to designate a query as a long query (string). The default value is '1'.

```
node[:mysql][:tunable][:long_query_time]
```

**max\_allowed\_packet**

The maximum allowed packet size (string). The default value is '32M'.

```
node[:mysql][:tunable][:max_allowed_packet]
```

**max\_connections**

The maximum number of concurrent client connections (string). The default value is '2048'.

```
node[:mysql][:tunable][:max_connections]
```

**max\_heap\_table\_size**

The maximum size of user-created MEMORY tables (string). The default value is '32M'.

```
node[:mysql][:tunable][:max_heap_table_size]
```

**net\_read\_timeout**

The amount of time, in seconds, to wait for more data from a connection (string). The default value is '30'.

```
node[:mysql][:tunable][:net_read_timeout]
```

**net\_write\_timeout**

The amount of time, in seconds, to wait for a block to be written to a connection (string). The default value is '30'.

```
node[:mysql][:tunable][:net_write_timeout]
```

**query\_cache\_limit**

The maximum size of an individual cached query (string). The default value is '2M'.

```
node[:mysql][:tunable][:query_cache_limit]
```

**query\_cache\_size**

The query cache size (string). The default value is '128M'.

```
node[:mysql][:tunable][:query_cache_size]
```

**query\_cache\_type**

The query cache type (string). The possible values are as follows:

- '0': No caching or retrieval of cached data.
- '1': Cache statements that don't begin with SELECT SQL\_NO\_CACHE.
- '2': Cache statements that begin with SELECT SQL\_CACHE.

The default value is '1'.

```
node[ :mysql][ :tunable][ :query_cache_type]
```

**thread\_cache\_size**

The number of client threads that are cached for re-use (string). The default value is '8'.

```
node[ :mysql][ :tunable][ :thread_cache_size]
```

**thread\_stack**

The stack size for each thread (string). The default value is '192K'.

```
node[ :mysql][ :tunable][ :thread_stack]
```

**wait\_timeout**

The amount of time, in seconds, to wait on a noninteractive connection. The default value is '180' (string).

```
node[ :mysql][ :tunable][ :wait_timeout]
```

**table\_cache**

The number of open tables (string). The default value is '2048'.

```
node[ :mysql][ :tunable][ :table_cache]
```

## nginx Attributes

The [nginx attributes](#) specify the Nginx configuration. For more information, see [Directive Index](#).

<a href="#">binary</a> (p. 415)	<a href="#">dir</a> (p. 415)	<a href="#">gzip</a> (p. 416)
<a href="#">gzip_comp_level</a> (p. 416)	<a href="#">gzip_disable</a> (p. 416)	<a href="#">gzip_http_version</a> (p. 416)
<a href="#">gzip_proxied</a> (p. 416)	<a href="#">gzip_static</a> (p. 416)	<a href="#">gzip_types</a> (p. 417)
<a href="#">gzip_vary</a> (p. 417)	<a href="#">keepalive</a> (p. 417)	<a href="#">keepalive_timeout</a> (p. 417)
<a href="#">log_dir</a> (p. 417)	<a href="#">user</a> (p. 417)	<a href="#">server_names_hash_bucket_size</a> (p.417)
<a href="#">worker_processes</a> (p. 417)	<a href="#">worker_connections</a> (p. 418)	

**binary**

The location of the Nginx binaries (string). The default value is '/usr/sbin/nginx'.

```
node[ :nginx][ :binary]
```

**dir**

The location of files such as configuration files (string). The default value is '/etc/nginx'.

```
node[ :nginx][ :dir]
```

### gzip

Whether gzip compression is enabled (string). The possible values are 'on' and 'off'. The default value is 'on'.

#### Caution

Compression can introduce security risks. To completely disable compression, set this attribute as follows:

```
node[:nginx][:gzip] = 'off'
```

```
node[:nginx][:gzip]
```

### gzip\_comp\_level

The compression level, which can range from 1–9, with 1 corresponding to the least compression (string). The default value is '2'.

```
node[:nginx][:gzip_comp_level]
```

### gzip\_disable

Disables gzip compression for specified user agents (string). The value is a regular expression and the default value is 'MSIE [1-6].(?!.\*SV1)'.

```
node[:nginx][:gzip_disable]
```

### gzip\_http\_version

Enables gzip compression for a specified HTTP version (string). The default value is '1.0'.

```
node[:nginx][:gzip_http_version]
```

### gzip\_proxied

Whether and how to compress the response to proxy requests, which can take one of the following values (string):

- 'off': do not compress proxied requests
- 'expired': compress if the Expire header prevents caching
- 'no-cache': compress if the Cache-Control header is set to "no-cache"
- 'no-store': compress if the Cache-Control header is set to "no-store"
- 'private': compress if the Cache-Control header is set to "private"
- 'no\_last\_modified': compress if Last-Modified is not set
- 'no\_etag': compress if the request lacks an ETag header
- 'auth': compress if the request includes an Authorization header
- 'any': compress all proxied requests

The default value is 'any'.

```
node[:nginx][:gzip_proxied]
```

### gzip\_static

Whether the gzip static module is enabled (string). The possible values are 'on' and 'off'. The default value is 'on'.

```
node[:nginx][:gzip_static]
```

**gzip\_types**

A list of MIME types to be compressed (list of string). The default value is [ 'text/plain', 'text/html', 'text/css', 'application/x-javascript', 'text/xml', 'application/xml', 'application/xml+rss', 'text/javascript' ].

```
node[:nginx][:gzip_types]
```

**gzip\_vary**

Whether to enable a Vary:Accept-Encoding response header (string). The possible values are 'on' and 'off'. The default value is 'on'.

```
node[:nginx][:gzip_vary]
```

**keepalive**

Whether to enable a keep-alive connection (string). The possible values are 'on' and 'off'. The default value is 'on'.

```
node[:nginx][:keepalive]
```

**keepalive\_timeout**

The maximum amount of time, in seconds, that a keep-alive connection remains open (number). The default value is 65.

```
node[:nginx][:keepalive_timeout]
```

**log\_dir**

The location of the log files (string). The default value is '/var/log/nginx'.

```
node[:nginx][:log_dir]
```

**user**

The user (string). The default values are as follows:

- Amazon: 'www-data'
- Ubuntu: 'nginx'

```
node[:nginx][:user]
```

**server\_names\_hash\_bucket\_size**

The bucket size for hash tables of server names, which can be set to 32, 64, or 128 (number). The default value is 64.

```
node[:nginx][:server_names_hash_bucket_size]
```

**worker\_processes**

The number of worker processes (number). The default value is 10.

```
node[:nginx][:worker_processes]
```

**worker\_connections**

The maximum number of worker connections (number). The default value is 1024. The maximum number of clients is set to `worker_processes * worker_connections`.

```
node[:nginx][:worker_connections]
```

## opsworks\_java Attributes

The `opsworks_java` attributes specify the Tomcat server configuration. For more information, see [Apache Tomcat Configuration Reference](#).

<a href="#">apache_tomcat_bind_mod</a> (p. 418)	<a href="#">apache_tomcat_bind_mod</a> (p. 418)	<a href="#">apache_tomcat_bind_path</a> (p. 418)
<a href="#">auto_deploy</a> (p. 419)	<a href="#">connection_timeout</a> (p. 419)	<a href="#">datasources</a> (p. 419)
<a href="#">java_app_server_version</a> (p. 419)	<a href="#">java_shared_lib_dir</a> (p. 419)	<a href="#">jvm_pkg Attributes</a> (p. 419)
<a href="#">custom_pkg_location_url_debian</a> (p. 419)	<a href="#">java_home_basedir</a> (p. 420)	<a href="#">custom_pkg_location_url_rhel</a> (p. 419)
<a href="#">use_custom_pkg_location</a> (p. 419)	<a href="#">jvm_options</a> (p. 420)	<a href="#">jvm_version</a> (p. 420)
<a href="#">mysql_connector_jar</a> (p. 420)	<a href="#">port</a> (p. 420)	<a href="#">secure_port</a> (p. 420)
<a href="#">shutdown_port</a> (p. 420)	<a href="#">threadpool_max_threads</a> (p. 420)	<a href="#">threadpool_min_spare_threads</a> (p. 420)
<a href="#">unpack_wars</a> (p. 421)	<a href="#">uri_encoding</a> (p. 421)	<a href="#">use_ssl_connector</a> (p. 421)
<a href="#">use_threadpool</a> (p. 421)	<a href="#">userdatabase_pathname</a> (p. 421)	

**ajp\_port**

The AJP port (number). The default value is 8009.

```
node['opsworks_java']['ajp_port']
```

**apache\_tomcat\_bind\_mod**

The proxy module (string). The default value is `proxy_http`. You can override this attribute to specify the AJP proxy module, `proxy_ajp`.

```
node['opsworks_java']['apache_tomcat_bind_mod']
```

**apache\_tomcat\_bind\_path**

The Apache-Tomcat bind path (string). The default value is `/`. You should not override this attribute; changing the bind path can cause the application to stop working.

```
node['opsworks_java']['apache_tomcat_bind_path']
```

**auto\_deploy**

Whether to autodeploy (Boolean). The default value is `true`.

```
node['opsworks_java']['auto_deploy']
```

**connection\_timeout**

The connection timeout, in milliseconds (number). The default value is 20000 (20 seconds).

```
node['opsworks_java']['connection_timeout']
```

**datasources**

A set of attributes that define JNDI resource names (string). For more information on how to use this attribute, see [Deploying a JSP App with a Database Connection \(p. 85\)](#). The default value is an empty hash, which can be filled with custom mappings between app short names and JNDI names. For more information, see [Deploying a JSP App with a Database Connection \(p. 85\)](#).

```
node['opsworks_java']['datasources']
```

**java\_app\_server\_version**

The Java app server version (number). The default value is 7. You can override this attribute to specify version 6. If you install a nondefault JDK, this attribute is ignored.

```
node['opsworks_java']['java_app_server_version']
```

**java\_shared\_lib\_dir**

The directory for the Java shared libraries (string). The default value is `/usr/share/java`.

```
node['opsworks_java']['java_shared_lib_dir']
```

**jvm\_pkg Attributes**

A set of attributes that you can override to install a nondefault JDK.

**use\_custom\_pkg\_location**

Whether to install a custom JDK instead of OpenJDK (Boolean). The default value is `false`.

```
node['opsworks_java']['jvm_pkg']['use_custom_pkg_location']
```

**custom\_pkg\_location\_url\_debian**

The location of the JDK package to be installed on Ubuntu instances (string). The default value is `'http://aws.amazon.com/'`, which is simply an initialization value with no proper meaning. If you want to install a nondefault JDK, you must override this attribute and set it to the appropriate URL.

```
node['opsworks_java']['jvm_pkg']['custom_pkg_location_url_debian']
```

**custom\_pkg\_location\_url\_rhel**

The location of the JDK package to be installed on Amazon Linux instances (string). The default value is `'http://aws.amazon.com/'`, which is simply an initialization value with no proper meaning. If you want to install a nondefault JDK, you must override this attribute and set it to the appropriate URL.

```
node['opsworks_java']['jvm_pkg']['custom_pkg_location_url_rhel']
```

**java\_home\_basedir**

The directory that the JDK package will be extracted to (string). The default value is /usr/local. You do not need to specify this setting for RPM packages; they include a complete directory structure.

```
node['opsworks_java']['jvm_pkg']['java_home_basedir']
```

**jvm\_options**

The JVM command line options, which allow you to specify settings such as the heap size (string). A common set of options is -Djava.awt.headless=true -Xmx128m -XX:+UseConcMarkSweepGC. The default value is no options.

```
node['opsworks_java']['jvm_options']
```

**jvm\_version**

The OpenJDK version (number). The default value is 7. You can override this attribute to specify OpenJDK version 6. If you install a nondefault JDK, this attribute is ignored.

```
node['opsworks_java']['jvm_version']
```

**mysql\_connector\_jar**

The MySQL connector library's JAR file (string). The default value is mysql-connector-java.jar.

```
node['opsworks_java']['mysql_connector_jar']
```

**port**

The standard port (number). The default value is 8080.

```
node['opsworks_java']['port']
```

**secure\_port**

The secure port (number). The default value is 8443.

```
node['opsworks_java']['secure_port']
```

**shutdown\_port**

The shutdown port (number). The default value is 8005.

```
node['opsworks_java']['shutdown_port']
```

**threadpool\_max\_threads**

The maximum number of threads in the thread pool (number). The default value is 150.

```
node['opsworks_java']['threadpool_max_threads']
```

**threadpool\_min\_spare\_threads**

The minimum number of spare threads in the thread pool (number). The default value is 4.

```
node['opsworks_java']['threadpool_min_spare_threads']
```

**unpack\_wars**

Whether to unpack WAR files (Boolean). The default value is `true`.

```
node['opsworks_java']['unpack_wars']
```

**uri\_encoding**

The URI encoding (string). The default value is `UTF-8`.

```
node['opsworks_java']['uri_encoding']
```

**use\_ssl\_connector**

Whether to use an SSL connector (Boolean). The default value is `false`.

```
node['opsworks_java']['use_ssl_connector']
```

**use\_threadpool**

Whether to use a thread pool (Boolean). The default value is `false`.

```
node['opsworks_java']['use_threadpool']
```

**userdatabase\_pathname**

The user database path name (string). The default value is `conf/tomcat-users.xml`.

```
node['opsworks_java']['userdatabase_pathname']
```

## passenger\_apache2 Attributes

The `passenger_apache2` attributes specify the [Phusion Passenger](#) configuration. For more information, see [Phusion Passenger users guide](#), [Apache version](#).

<a href="#">friendly_error_pages (p. 421)</a>	<a href="#">gem_bin (p. 421)</a>	<a href="#">gems_path (p. 422)</a>
<a href="#">high_performance_mode (p. 422)</a>	<a href="#">root_path (p. 422)</a>	<a href="#">max_instances_per_app (p. 422)</a>
<a href="#">max_pool_size (p. 422)</a>	<a href="#">max_requests (p. 422)</a>	<a href="#">module_path (p. 422)</a>
<a href="#">pool_idle_time (p. 422)</a>	<a href="#">rails_app_spawner_idle_time (p. 423)</a>	<a href="#">rails_framework_spawner_idle_time (p. 423)</a>
<a href="#">rails_spawn_method (p. 423)</a>	<a href="#">ruby_bin (p. 423)</a>	<a href="#">ruby_wrapper_bin (p. 423)</a>
<a href="#">stat_throttle_rate (p. 423)</a>	<a href="#">version (p. 423)</a>	

**friendly\_error\_pages**

Whether to display a friendly error page if an application fails to start (string). This attribute can be set to 'on' or 'off'; the default value is 'off'.

```
node[:passenger][:friendly_error_pages]
```

**gem\_bin**

The location of the Gem binaries (string). The default value is `'/usr/local/bin/gem'`.

```
node[:passenger][:gem_bin]
```

#### **gems\_path**

The gems path (string). The default value depends on the Ruby version. For example:

- Ruby version 1.8: '/usr/local/lib/ruby/gems/1.8/gems'
- Ruby version 1.9: '/usr/local/lib/ruby/gems/1.9.1/gems'

```
node[:passenger][:gems_path]
```

#### **high\_performance\_mode**

Whether to use Passenger's high-performance mode (string). The possible values are 'on' and 'off'. The default value is 'off'.

```
node[:passenger][:high_performance_mode ]
```

#### **root\_path**

The Passenger root directory (string). The default value depends on the Ruby and Passenger versions. In Chef syntax, the value is "#{node[:passenger][:gems\_path]}/passenger-#{passenger[:version]}".

```
node[:passenger][:root_path]
```

#### **max\_instances\_per\_app**

The maximum number of application processes per app (number). The default value is 0. For more information, see [PassengerMaxInstancesPerApp](#).

```
node[:passenger][:max_instances_per_app]
```

#### **max\_pool\_size**

The maximum number of application processors (number). The default value is 8. For more information, see [PassengerMaxPoolSize](#).

```
node[:passenger][:max_pool_size]
```

#### **max\_requests**

The maximum number of requests (number). The default value is 0.

```
node[:passenger][:max_requests]
```

#### **module\_path**

The module path (string). The default values are as follows:

- Amazon: "#{node['apache']['libexecdir (p. 402)']}/mod\_passenger.so"
- Ubuntu: "#{passenger[:root\_path (p. 422)]}/ext/apache2/mod\_passenger.so"

```
node[:passenger][:module_path]
```

#### **pool\_idle\_time**

The maximum time, in seconds, that an application process can be idle (number). The default value is 14400 (4 hours). For more information, see [PassengerPoolIdleTime](#).

```
node[:passenger][:pool_idle_time]
```

#### **rails\_app\_spawner\_idle\_time**

The maximum idle time for the Rails app spawner (number). If this attribute is set to zero, the app spawner does not time out. The default value is 0. For more information, see [Spawning Methods Explained](#).

```
node[:passenger][:rails_app_spawner_idle_time]
```

#### **rails\_framework\_spawner\_idle\_time**

The maximum idle time for the Rails framework spawner (number). If this attribute is set to zero, the framework spawner does not time out. The default value is 0. For more information, see [Spawning Methods Explained](#).

```
node[:passenger][:rails_framework_spawner_idle_time]
```

#### **rails\_spawn\_method**

The Rails spawn method (string). The default value is 'smart-lv2'. For more information, see [Spawning Methods Explained](#).

```
node[:passenger][:rails_spawn_method]
```

#### **ruby\_bin**

The location of the Ruby binaries (string). The default value is '/usr/local/bin/ruby'.

```
node[:passenger][:ruby_bin]
```

#### **ruby\_wrapper\_bin**

The location of the Ruby wrapper script (string). The default value is '/usr/local/bin/ruby\_gc\_wrapper.sh'.

```
node[:passenger][:ruby_wrapper_bin]
```

#### **stat\_throttle\_rate**

The rate at which Passenger performs file system checks (number). The default value is 5, which means that the checks will be performed at most once every 5 seconds. For more information, see [PassengerStatThrottleRate](#).

```
node[:passenger][:stat_throttle_rate]
```

#### **version**

The version (string). The default value is '3.0.9'.

```
node[:passenger][:version]
```

## **ruby Attributes**

The [ruby attributes](#) specify the Ruby configuration. Note that attribute usage changed with the introduction of semantic versioning in Ruby 2.1.

**full\_version**

The full version number (string).

```
[ :ruby] [:full_version]
```

**major\_version**

The major version number (string).

```
[ :ruby] [:major_version]
```

**minor\_version**

The minor version number (string). For Chef 11.4 stacks, this attribute is valid for Ruby version 2.1 and later. For earlier Ruby versions, use the `pkgrelease` attribute.

```
[ :ruby] [:minor_version]
```

**patch**

The patch number (string). This attribute is valid for Ruby version 2.0.0 and earlier. For later Ruby versions, use the `patch_version` attribute.

```
[ :ruby] [:patch]
```

The patch number must be prefaced by `p`. For example, you would use the following custom JSON to specify patch level 484.

```
{
  "ruby": { "patch": "p484" }
}
```

**patch\_version**

The patch version (string). This attribute is valid for Ruby version 2.1 and later. For earlier Ruby versions, use the `patch` attribute.

```
[ :ruby] [:patch_version]
```

**pkgrelease**

The package release number (string).

```
[ :ruby] [:pkgrelease]
```

## unicorn Attributes

The `unicorn` attributes specify the `Unicorn` configuration. For more information, see `Unicorn::Configurator`.

<a href="#">accept_filter (p. 425)</a>	<a href="#">backlog (p. 425)</a>	<a href="#">delay (p. 425)</a>
<a href="#">tcp_nodelay (p. 425)</a>	<a href="#">tcp_nopush (p. 425)</a>	<a href="#">preload_app (p. 425)</a>

<a href="#">timeout (p. 425)</a>	<a href="#">tries (p. 425)</a>	<a href="#">version (p. 425)</a>
<a href="#">worker_processes (p. 426)</a>		

**accept\_filter**

The accept filter, 'httpready' or 'dataready' (string). The default value is 'httpready'.

```
node[:unicorn][:accept_filter]
```

**backlog**

The maximum number of requests that the queue can hold (number). The default value is 1024.

```
node[:unicorn][:backlog]
```

**delay**

The amount of time, in seconds, to wait to retry binding a socket (number). The default value is 0.5.

```
node[:unicorn][:delay]
```

**preload\_app**

Whether to preload an app before forking a worker process (Boolean). The default value is true.

```
node[:unicorn][:preload_app]
```

**tcp\_nodelay**

Whether to disable Nagle's algorithm for TCP sockets (Boolean). The default value is true.

```
node[:unicorn][:tcp_nodelay]
```

**tcp\_nopush**

Whether to enable TCP\_CORK (Boolean). The default value is false.

```
node[:unicorn][:tcp_nopush]
```

**timeout**

The maximum amount time, in seconds, that a worker is allowed to use for each request (number). Workers that exceed the timeout value are terminated. The default value is 60.

```
node[:unicorn][:timeout]
```

**tries**

The maximum number of times to retry binding to a socket (number). The default value is 5.

```
node[:unicorn][:tries]
```

**version**

The Unicorn version (string). The default value is '4.7.0'.

```
node[:unicorn][:version]
```

**worker\_processes**

The number of worker processes (number). The default value is `max_pool_size`, if it exists, and 4 otherwise.

```
node[ :unicorn] [ :worker_processes]
```

# AWS OpsWorks Resources

---

The following related resources can help you as you work with this service.

## Reference Guides, Tools, and Support Resources

Several helpful guides, forums, contact info, and other resources are available from AWS OpsWorks and Amazon Web Services.

- [AWS OpsWorks API Reference](#) – Descriptions, syntax, and usage examples about AWS OpsWorks actions and data types, including common parameters and error codes.
- [AWS OpsWorks Technical FAQ](#) – Top questions developers have asked about this product.
- [AWS OpsWorks Release Notes](#) – A high-level overview of the current release. This document specifically notes any new features, corrections, and known issues.
- [AWS Tools for PowerShell](#) – A set of Windows PowerShell cmdlets that expose the functionality of the AWS SDK for .NET in the PowerShell environment.
- [AWS Command Line Interface](#) – A uniform command line syntax for accessing AWS services. The AWS CLI uses a single setup process to enable access for all supported services.
- [AWS OpsWorks Command Line Reference](#) – AWS OpsWorks-specific commands for use at a command line prompt.
- [AWS Developer Tools](#) – Links to developer tools and resources that provide documentation, code samples, release notes, and other information to help you build innovative applications with AWS.
- [AWS Support Center](#) – This site brings together information about your recent support cases and results from AWS Trusted Advisor and health checks, as well as providing links to discussion forums, technical FAQs, the service health dashboard, and information about AWS support plans.
- [AWS Premium Support Information](#) – The primary web page for information about AWS Premium Support, a one-on-one, fast-response support channel to help you build and run applications on AWS Infrastructure Services.
- [Contact Us](#) – Links for inquiring about your billing or account. For technical questions, use the discussion forums or support links above.
- [Terms of Use](#) – Detailed information about our copyright and trademark; your account, license, and site access; and other topics.

# AWS Software Development Kits

Amazon Web Services provides software development kits for accessing AWS OpsWorks from several different programming languages. The SDK libraries automate a number of common tasks, including cryptographically signing your service requests, retrying requests, or handling error responses.

- **AWS SDK for Java** – [Setup](#) and [other documentation](#)
- **AWS SDK for .NET** – [Setup](#) and [other documentation](#)
- **AWS SDK for PHP** – [Setup](#) and [other documentation](#)
- **AWS SDK for Ruby** – [Setup](#) and [other documentation](#)
- **AWS SDK for JavaScript in Node.js** – [Setup](#) and [other documentation](#)
- **AWS SDK for Python (Boto)** – [Setup](#) and [other documentation](#)

# History

---

The following table describes the important changes to the documentation in this release of AWS OpsWorks.

- **API version:** 2013-02-18
- **Latest documentation update:** Aug 11, 2014

Change	Description	Date Changed
New Feature	Added support for defining environment variables on application servers.	Jul 16, 2014
New Documentation	Added Cookbooks 101, a tutorial introduction to implementing cookbooks.	Jul 16, 2014
New Feature	Added support for CloudTrail.	Jun 4, 2014
New Feature	Added support for Amazon RDS.	May 14, 2014
New Feature	Added support for Chef 11.10 and Berkshelf.	Mar 27, 2014
New Feature	Added support for Amazon EBS PIOPS volumes.	Dec 16, 2013
New Feature	Added resource-based permissions.	Dec 5, 2013
New Feature	Added resource management.	Oct 7, 2013
New Feature	Added support for VPCs.	August 29, 2013
New Features	Added support for custom AMIs and Chef 11.4.	July 24, 2013
New Feature	Added console support for multiple layers per instance.	July 1, 2013
New Features	Added support for Amazon EBS-backed instances, Elastic Load Balancing, and Amazon CloudWatch monitoring.	May 14, 2013
Initial Release	Initial release of the AWS OpsWorks User Guide.	February 18, 2013