Bootstrapping Applications via AWS CloudFormation

With AWS CloudFormation, you write templates to define the set of resources that you need to run your applications. These resources might include Amazon Elastic Cloud Compute (EC2) instances, Amazon Relational Database Service (RDS) instances, and Elastic Load Balancing load balancers. The template describes *what* resources you need and AWS CloudFormation provisions those resources in an orderly and predictable fashion, creates them in parallel where possible, and deals with any failures or transient issues. For more information about using AWS CloudFormation see the AWS CloudFormation product detail page.

Although AWS CloudFormation takes care of provisioning all the resources, you still must deploy, configure, and run (bootstrap) your applications on an Amazon EC2 instance. You have many options for bootstrapping your applications, each of which affects how quickly your application is ready and how flexible your deployment methods can be. The following sections describe how AWS CloudFormation can help you to configure and install your application on Amazon EC2 instances.

Overview

Before describing the details, a brief overview of various techniques will show you the options and help you understand the trade-offs for each solution.

Using CloudInit and Amazon EC2 User Data to Customize an Instance at Launch Time

With CloudInit, you can pass executable actions to instances at launch time through the Amazon EC2 user-data fields so you can configure and customize the instances at launch time. Both the Ubuntu Linux AMI and the Amazon Linux AMI contain a version of CloudInit. Dynamically configuring an AMI at startup means you can use a common base AMI for different use cases.

By using CloudInit to execute scripts at launch time, you can download and install software without having to have the applications preinstalled in the AMI. For more details about CloudInit, see the following resources:

- Amazon Linux AMI Basics
- Ubuntu CloudInit documentation

Because applications are installed and configured at launch time, the applications can take longer to get up and running. In some cases, such as scaling out an application with Auto Scaling, the additional time to have an instance in service might be problematic because they are launched when you need to meet additional demand. To speed up the instance launch, you can use a hybrid solution where some of the core, stable application components are built into the AMI (such as base infrastructure services) and components that change more frequently are installed at launch time. This solution gives you the flexibility to easily upgrade the application version on new instances. By combining AMI deployments with CloudInit, you have control of the trade-offs between instance launch time and flexibility to change application components.

Note: Before you use CloudInit, consider the following constraints:

- The CloudInit script is passed by using the Amazon EC2 user data field. This field has a size limit of 16K bytes.
- Because CloudInit is script-based, it is nondeclarative, making it challenging to audit exactly what is installed.

The Windows AMIs that are provided by Amazon Web Services have a similar mechanism to CloudInit that allows you to run scripts at instance launch. For more details see Configuring a Windows Instance Using the EC2Config Service.

Using AWS CloudFormation Metadata and Helper Scripts

You can use AWS CloudFormation templates to define the set of packages, files, and operating system services that are installed on EC2 instances in a declarative way. AWS CloudFormation helper scripts run on the instance to interpret the definitions in your template, installing packages, creating files, and starting or restarting services on the instance.

The AWS CloudFormation helper scripts build on the basic CloudInit functionality so that you can create a common, simple CloudInit startup script that is driven from the template. You describe *what* applications need to be installed on the host in the template and AWS CloudFormation takes care of *how* they are installed.

In later sections, you'll see how to define packages, files, and other artifacts in your template. You'll also see a simple CloudInit script for installing your applications at instance launch time.

Other Strategies

Baking Your Application into an Amazon Machine Image (AMI)

Amazon Web Services (AWS) publishes many Amazon Machine Images (AMIs) that contain common software configurations for public use. In addition, the AWS developer community has published many custom AMIs. You can also create your own custom AMIs so that you can quickly and easily start new instances that have everything you need for your application. For example, if your application is a website or a web service, your AMI could include a web server, the associated static content, and the code for the dynamic pages. After you launch an instance with this AMI, your application is running and ready to accept requests.

Amazon EC2 provides an instance metadata store that can be accessed locally from the instance. You can use the data stored in the instance metadata store to further customize AMIs while the EC2 instance is being launched. For example, the database connection string can be passed to the EC2 instance through the instance metadata store, enabling the same AMI to be used by multiple applications that access different Amazon RDS database instances.

The EC2 user guide shows you how to create and manage AMIs.

AMIs provide you with a way to create *golden images* containing a well-defined set of OS and application versions as well as files and other configurations that you can use over and over, so that each instance has the same software installed and is configured in the same way. In addition, all of the software is preinstalled and ready as soon as the instance launches, making AMI-based deployments fast.

With AMI-based deployments, each time you want to upgrade your applications, apply patches, or change configuration settings that are baked into the AMI, you need to create a new AMI. Because

of this, you must define a strategy for versioning your AMIs and operational processes to ensure new EC2 instances are launched with your latest AMI version.

Integration with Chef and Puppet

Chef is an open source infrastructure automation solution, written in Ruby. Chef automates the configuration of your systems and the applications that sit on top of it. It is a client-server application where clients pull the configuration from the Chef server, and all the work to transform the configuration into an instance that serves a function takes place on the instance itself.

Puppet is an Open Source IT services platform developed under the guidance of the team at Puppet Labs. With Puppet, you can manage the provisioning, patching, and configuration of operating systems and their application stacks efficiently as your infrastructure grows. Puppet is a client-server application where clients pull configuration from a puppet master.

By building on the ability to define packages, files, and configurations in an AWS CloudFormation template, you can use AWS CloudFormation to do the following :

1. Bootstrap your EC2 instances with the Chef or Puppet client software
2. Deploy and configure a Chef server or a puppet master

In addition to the basic bootstrapping, AWS CloudFormation provides plug-ins to the Chef and Puppet client software so you can define configuration keys in the template metadata for bootstrapping applications (such as the host name of the database to connect to). This provides an easy, one-stop-shop to define the role for a server and any runtime properties that the role might require.

You can find more details of how to integrate with Chef and Puppet in the following whitepapers:

- Integrating AWS CloudFormation with Chef
- Integrating AWS CloudFormation with Puppet

Customizing Instances with CloudInit and EC2 User Data

When you launch an instance using an Amazon Machine Image (AMI), you can pass information to the AMI to customize it at instance launch time. Amazon EC2 provides a *user data* field as part of the Amazon EC2 instance metadata store that can be populated at instance launch. The user data is passed as a base64 encoded string and is available locally on the instance when the instance is running.

AWS CloudFormation also provides a number of intrinsic functions such as *Fn::Base64* and *Fn::Join* along with *Ref* and *Fn::GetAtt* that you can use to construct strings that get passed to the Amazon EC2 user data. The actual format of the string is entirely up to you and your application. The format can be key-value pairs in a form like this:

```
Key1=Value1;Key2=Value2;Key3=Value3
Key1=Value1&Key2=Value2&Key3=Value3
{Key1=>Value1, Key2=>Value2, Key3=>Value3}
```

The following snippet is an example of a configuration string for a database:

```
Database=Foo:3306&DBUser=admin&DBPassword=admin&WSPort=80
```

```
"MyInstance": {
  "Type": "AWS::EC2::Instance",
  "Properties": {
    :
    "UserData": {
      "Fn::Base64": { "Fn::Join": [ "", [
        "Database=", {"Fn::GetAtt": ["SampleDB", "Endpoint.Address"]}, ":", {"Ref": "DatabasePort"}, "&",
        "DBUser=", {"Ref": "DatabaseUser"}, "&",
        "DBPassword=", {"Ref": "DatabasePassword"}, "&",
        "WSPort=", {"Ref": "WebServerPort"}
      ]]}}
    }
  }
}
```

The instance must have a script that retrieves the string from the Amazon EC2 metadata store and acts on it. The user data can be retrieved by using `wget` or an equivalent command from the Amazon EC2 metadata store on the instance at `http://169.254.169.254/latest/user-data`. Note that the metadata store is accessed by using a URL at a constant IP address; the metadata store is always local and accessible at this specific IP address, no matter what the platform—Linux, Windows, etc.

You can also use a helper script on Amazon Linux AMIs (`/opt/aws/bin/ec2-metadata`) to extract the user data and any other metadata from the metadata store.

```
$ /opt/aws/bin/ec2-metadata --help
ec2-metadata v0.1.2
Use to retrieve EC2 instance metadata from within a running EC2 instance.
e.g., to retrieve instance ID: ec2-metadata -i
            to retrieve AMI ID: ec2-metadata -a
            to get help: ec2-metadata --help
For more information on Amazon EC2 instance meta-data, refer to the documentation at
http://docs.amazonwebservices.com/AWSEC2/2008-05-05/DeveloperGuide/AESDG-chapter-instancedata.html

Usage: ec2-metadata <option>
Options:
--all                    Show all metadata information for this host (also default).
-a/--ami-id              The AMI ID used to launch this instance.
-l/--ami-launch-index    The index of this instance in the reservation (per AMI).
-m/--ami-manifest-path   The manifest path of the AMI with which the instance was launched.
-n/--ancestor-ami-ids    The AMI IDs of any instances that were rebundled to create this AMI.
-b/--block-device-mapping Defines native device names to use when exposing virtual devices.
-i/--instance-id         The ID of this instance
-t/--instance-type       The type of instance to launch. For more information, see Instance Types.
-h/--local-hostname      The local hostname of the instance.
-o/--local-ipv4          Public IP address if launched with direct addressing; private IP address if
launched with public addressing.
-k/--kernel-id           The ID of the kernel launched with this instance, if applicable.
-z/--availability-zone   The Availability Zone in which the instance launched. Same as placement
-c/--product-codes       Product codes associated with this instance.
-p/--public-hostname     The public host name of the instance.
-v/--public-ipv4         NATted public IP Address
-u/--public-keys         Public keys. Only available if supplied at instance launch time
-r/--ramdisk-id          The ID of the RAM disk launched with this instance, if applicable.
-e/--reservation-id      ID of the reservation.
-s/--security-groups     Names of the security groups the instance is launched in. Only available if
```

```
supplied at instance launch time
-d/--user-data            User-supplied data. Only available if supplied at instance launch time.
```

For details on using the Amazon EC2 metadata store, see the Amazon EC2 user guide.

For many Linux AMIs, you can add your scripts to the */etc/rc.local* boot script, which would run your configuration scripts when the instance is launched as the *root* user. In general, you should check out the boot details for your specific operating system build to figure out where to deploy your boot script and what credentials the script uses.

The user data field is immutable. After an instance is launched, you cannot modify the user data field, which might affect the configuration information that you want it to convey. To work around this, try passing the location of an Amazon SimpleDB domain or an Amazon Simple Storage Service (S3) bucket through the user data rather than the data itself. Doing so lets you make changes to the actual data in the data store without having to change the location of the configuration information.

The downside of using other stores is that you will need to make sure a valid set of user credentials is available on the instance. You can pass credentials to the instance in user data; however, we recommend that if you do this, you pass the AWS access key ID and secret key of an AWS Identity and Access Management (IAM) user with limited permissions. You should grant the IAM user only the read access permissions it needs to retrieve the configuration information.

You can use AWS CloudFormation to create IAM users, groups, and policies. From a within a single template, you can create a user, set appropriate polices, create an access key ID and secret key pair, and then add the credentials to the instance through the user data. By adding the IAM user in the template, you have a user whose existence is tied to the lifetime of the stack, with each new stack having a separate, unique user.

Using AWS CloudFormation Metadata and Helpers

In the AWS CloudFormation template metadata, you can attach metadata to any resource. Any intrinsic functions (such as `Fn::GetAtt` and `Ref`) in the metadata are resolved by AWS CloudFormation when you create a stack. Applications can retrieve the metadata by using the `aws cloudformation get-stack-resource` command line tool or the `GetStackResource` API call, specifying the stack name and the logical resource name.

The following example of the uses the AWS CloudFormation metadata to define an application configuration:

```
"Ec2Instance" : {
  "Type" : "AWS::EC2::Instance",
  "Metadata" : {
    "Comment" : "This metadata is available via the DescribeStackResource API call",
    "QueueSize" : "30",
    "CName" : { "Fn::GetAtt": [ "MyLoadBalancer", "DNSName" ] },
    "RDSConnectionString" : { "Fn::Join" : [ "", [
         "jdbc:mysql://",
         { "Fn::GetAtt" : [ "MyDB", "Endpoint.Address" ] }, ":",
         { "Fn::GetAtt" : [ "MyDB", "Endpoint.Port" ] }, "/",
         { "Ref" : "DatabaseName" }]]}
```

```
  },
  :
}
```

In this example, the metadata defines four key-value pairs:

- `Comment` is a text value that you can use in the metadata to insert resource-specific comments into the template.
- `QueueSize` is a numeric value.
- `CName` is a string value whose value is the DNSName attribute of the MyLoadBalancer Elastic Load Balancing load balancer that is defined elsewhere in the template.
- `RDSConnectionString` is a string value that is constructed by using the `Fn::Join`, `Ref`, and `Fn::GetAtt` intrinsic functions to construct a JDBC connection string from an RDS instance and the `DatabaseName` parameter defined elsewhere in the template.

By using AWS CloudFormation metadata, you can pass configuration values to applications running in an Amazon EC2 instance in much the same way as using the Amazon EC2 user data field. Unlike the immutable user data field, the resource metadata can be updated dynamically by using the `aws cloudformation update-stack` command line tool or the `UpdateStack` API.

Note: Before you use the template metadata, consider the following constraints:

- You must have valid AWS credentials on the Amazon EC2 instance to make the `DescribeStackResource` API call or use the `aws cloudformation describe-stack-resource` CLI command.
- The metadata is stored by the AWS CloudFormation service and is not locally stored on the instance. In other words, the bootstrapping sequence depends on your instance's connection to the AWS CloudFormation service.

In order to read the metadata from an Amazon EC2 instance that is created in the template, we recommend using an IAM role that is passed to the instance at launch by using an instance profile. For more information on IAM roles and EC2 instance profiles, see Granting Applications that Run on Amazon EC2 Instances Access to AWS Resources in the *AWS Identity and Access Management User Guide*.

The following example creates an IAM role and passes it to the Amazon EC2 instance through the EC2 instance profile.

```
Resources: {

  myEC2Instance: {
    Type: "AWS::EC2::Instance",
    Properties: {
      :
      IamInstanceProfile: { Ref: "RootInstanceProfile" },
      :
    }
  },
```

```
  RootRole: {
    Type: "AWS::IAM::Role",
    Properties: {
      AssumeRolePolicyDocument: {
        Statement: [{
          Effect: "Allow",
          Principal: { Service: ["ec2.amazonaws.com" ] },
          Action: [ "sts:AssumeRole" ]
      }]},
      Path: "/"
    }
  },

  RolePolicy: {
    Type: "AWS::IAM::Policy",
    Properties: {
      PolicyName: "root",
      PolicyDocument: {
        Statement: [{
          Effect: "Allow",
          Action: "cloudformation:DescribeStackResource",
          Resource: "*"
      }]},
      Roles: [{ Ref: "RootRole" }]
    }
  },

  RootInstanceProfile: {
    Type: "AWS::IAM::InstanceProfile",
    Properties: {
      Path: "/",
      Roles: [{ Ref: "RootRole" }]
    }
  }
:
}
```

In the previous example, we created an IAM role that has a policy that grants permission to call the AWS CloudFormation `DescribeStackResource` API only. If you use the AWS Command Line Interface or the AWS SDKs from the EC2 instance to access the metadata in the template, temporary credentials that are associated with the IAM role are automatically used in the call to the AWS CloudFormation service.

Your application might also require access to other AWS services when it is running. Because only one IAM role can be associated with an EC2 instance when you use an instance profile, you need to grant additional permissions to the role in the template, depending on your application needs.

Installing Packages and Files Using AWS CloudFormation Helpers

If you are installing and configuring your applications on Amazon EC2 dynamically at instance launch time, you typically need to pull and install packages, deploy files, and ensure that services are started. AWS CloudFormation provides a set of helper scripts (written in Python) that, in conjunction with the resource metadata you defined in the template, can be used to install software and start services. These helper scripts are run on the Amazon EC2 instance.

Currently, AWS CloudFormation provides the following helper scripts:

- ***cfn-init***: Retrieves and interprets the resource metadata to install packages, create files, and start services.
- ***cfn-signal***: A simple wrapper to signal an AWS CloudFormation `WaitCondition` for synchronizing other resources in the stack when the application is ready.
- ***cfn-get-metadata***: A wrapper script that retrieves either all metadata that is defined for a resource or path to a specific key or a subtree of the resource metadata.
- ***cfn-hup***: A daemon that checks for updates to metadata and executes custom hooks when changes are detected.

These scripts are installed by default on Amazon Linux AMIs in the */opt/aws/bin* directory. The scripts are also available through RPM and source for other Linux/Unix distributions. These scripts are installed on Amazon Windows AMIs by default. If you create your own Windows image for use with CloudFormation, see Configuring a Windows Instance Using EC2ConfigService in the *Amazon EC2 Microsoft Windows Guide* for instructions. You must set up a Windows instance with EC2ConfigService to work with the AWS CloudFormation bootstrapping tools.

The scripts are not executed by default. The following sections show you how to set up the scripts and execute them when your instances are launched.

Metadata Block Format

The `cfn-init` script uses the resource metadata that is defined in the `AWS::CloudFormation::Init` metadata key. The configuration is separated into a number of sections for archives, packages, files, and services. The following example shows how you can attach metadata for `cfn-init` to an Amazon EC2 instance resource within a template:

```
"Resources": {
  "MyInstance": {
    "Type": "AWS::EC2::Instance",
    "Metadata" : {
      "AWS::CloudFormation::Init" : {
        "config" : {
          "packages" : {
            :
          },
          "users" : {
            :
          },
          "groups" : {
            :
          },
          "sources" : {
            :
          },
```

```
          "files" : {
            :
          },
          "commands" : {
            :
          },
          "services" : {
            :
          }
        }
      }
    },
    "Properties": {
      :
    }
  }
}
```

The configuration sections are executed in the following order:

- The `Packages` section downloads and installs prepackaged applications and components. The `cfn-init` script currently supports the following package formats: apt, yum, rubygems, python, rpm, and MSI (on Windows).
- The `Users` section creates user accounts on the local OS. (Not supported on Windows.)
- The `Groups` section creates user groups on the local OS. (Not supported on Windows.)
- The `Sources` section downloads an archive file and unpacks it in a target directory on the EC2 instance. The following formats are supported: `tar`, `tar+gzip`, `tar+bz2`, and `zip`.
- The `Files` section creates arbitrary files by using the Files option. The content can be inline from the template or the content can be pulled from a URL.
- The `Commands` section executes arbitrary commands on the instance.
- The `Services` section defines which services should be enabled or disabled when the instance is launched.

Each of these sections is described in detail with example configurations in the AWS::CloudFormation::Init section of the *AWS CloudFormation User Guide*. For more details about Windows, see Bootstrapping AWS CloudFormation Windows Stacks.

Running cfn-init

The `cfn-init` helper script interprets the metadata that contains the sources, packages, files, and services. You run the script on the EC2 instance when it is launched. The script is installed by default on Amazon Linux and Windows AMIs.

If you are running other Linux AMIs, you must install the AWS CloudFormation helper scripts. Because these scripts need to run at boot time, you need to install them by using CloudInit (if your OS has CloudInit support) or bake an AMI with the OS and the CloudFormation helper scripts installed.

For more information about `cfn-init`, see `cfn-init` in the *AWS CloudFormation User Guide*.

The `cfn-init` helper script uses the `DescribeStackResource` API call to get the metadata from the AWS CloudFormation service. We strongly recommend that you use IAM roles and EC2

instance profiles to set credentials on the instance. Do not use the `--access-key`, `--secret-key`, `--role` or the `--credential-file` options for `cfn-init`. These are supported for backward compatibility only.

Because the metadata can be attached to any of the resources in the template, the `cfn-init` script takes the stack name (or stack ID) and the logical resource name that contains the metadata you want to use.

We recommend that you put your entire instance configuration into a single metadata property and run a single `cfn-init` command to configure the instance. You can create a single bootstrap script with which you can install any application. The following example shows a simple bootstrap script for an Amazon Linux AMI:

```
#!/bin/bash
/opt/aws/bin/cfn-init –-stack <stackname> --resource <resourcename> --region <region>
```

Using the intrinsic functions, you can inject this script into the base Amazon Linux AMI via the user data field, as shown in the following example (the bootstrap script runs as *root*):

```
"MyInstance": {
  "Type": "AWS::EC2::Instance",
  "Metadata": {
     :
  },
  "Properties": {
    :
    "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
      "#!/bin/bash\n",
      "/opt/aws/bin/cfn-init ",
      "         --stack ", { "Ref" : "AWS::StackName" },
      "         --resource MyInstance ",
      "         --region ", { "Ref" : "AWS::Region" }, "\n"
    ]]}}
  }
}
```

We recommend you use the latest version of the AWS CloudFormation bootstrap scripts when you launch an Amazon Linux AMI. To ensure that happens, add a `yum update` command to the user data script, as shown in the following example:

```
"MyInstance": {
  "Type": "AWS::EC2::Instance",
  "Metadata": {
     :
  },
  "Properties": {
    :
    "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
      "#!/bin/bash\n",
      "yum update -y aws-cfn-bootstrap\n",
      "/opt/aws/bin/cfn-init ",
      "         --stack ", { "Ref" : "AWS::StackName" },
      "         --resource MyInstance ",
      "         --region ", { "Ref" : "AWS::Region" }, "\n"
    ]]}}
  }
```

```
}
```

The following sample is a bootstrapping example on Windows:

```
"MyInstance": {
  "Type": "AWS::EC2::Instance",
  "Metadata": {
     :
  },
  "Properties": {
    :
    "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
      "<script>\n",
      "cfn-init.exe ",
      "         --stack ", { "Ref" : "AWS::StackName" },
      "         --resource MyInstance ",
      "         --region ", { "Ref" : "AWS::Region" }, "\n"
      "</script>\n"
    ]]}}
  }
}
```

In the previous example, the bootstrap script picks up the metadata from the Amazon EC2 instance. The script uses the pseudo parameters `AWS::StackName` and `AWS::Region` to get the stack name and the region in which the stack is created.

If you are not using the Amazon Linux AMI, Ubuntu, or Windows, you can execute the bootstrap script through standard techniques for the specific operating system. For example, on operating systems that support `rc.local`, you can append the bootstrap script to the `rc.local` startup script so that the script is executed on boot.

Waiting for Applications to Be *Ready* Before Continuing

In some cases, you might need to wait for an application to be deployed and running on one or more EC2 instances before the stack creation process can continue or be completed. To provide synchronization between the helper scripts on the instance and other resources in the template, AWS CloudFormation supports a `WaitCondition` resource. You can think of a wait condition as a timed semaphore. When the `WaitCondition` resource is created, a timer is started. The `WaitCondition` becomes CREATE_COMPLETE after it receives a signal. If no signal is received before the timeout, the `WaitCondition` enters the CREATE_FAILED state and the stack creation is rolled back.

To wait for the application to be ready, we can extend the previous template to create a `WaitCondition` and a `WaitConditionHandle`. Then use the `cfn-signal` helper script to signal that the application is installed. The `cfn-signal` script is a wrapper that makes it easy to signal a wait object from your bootstrap script.

For more information about `cfn-signal`, see `cfn-signal` in the *AWS CloudFormation User Guide*.

Extending the previous example, the following example creates a `WaitCondition` and a `WaitConditionHandle`. The `cfn-signal` is added to the bootstrap script.

```
"MyInstance": {
```

```
  "Type": "AWS::EC2::Instance",
  "Metadata": {
     :
  },
  "Properties": {
    :
    "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
      "#!/bin/bash\n",
      "yum update -y aws-cfn-bootstrap\n",
      "/opt/aws/bin/cfn-init ",
      "         --stack ", { "Ref" : "AWS::StackName" },
      "         --resource MyInstance ",
      "         --region ", { "Ref" : "AWS::Region" }, "\n"
      "/opt/aws/bin/cfn-signal -e $? '", { "Ref" : "MyHandle" }, "'\n",
    ]]}}
  }
},

"MyHandle" : {
  "Type" : "AWS::CloudFormation::WaitConditionHandle"
},

"MyWaitCondition" : {
  "Type" : "AWS::CloudFormation::WaitCondition",
  "DependsOn" : "MyInstance",
  "Properties" : {
    "Handle" : {"Ref" : "MyHandle"},
    "Timeout" : "600"
  }
}
```

In the previous example, the `WaitConditionHandle` is signaled with the return status from `cfn-init` (using the `$?` shell construct). By using a wait condition, if the application installation fails, the `WaitCondition` creation fails and the stack is rolled back.

Note the `DependsOn` property in the `WaitCondition`. The `DependsOn` property ensures that the `WaitCondition` is created after the EC2 instance is launched and guarantees that the timer that is associated with the `WaitCondition` does not start until the instance is running and the configuration scripts can run. Without the `DependsOn`, the `WaitCondition` could be created at any time during the stack creation process, making the timer nondeterministic. (AWS CloudFormation picks the optimal way to start resources based on a dependency tree.)

The following example shows how to add a signal for Windows:

```
"MyInstance": {
  "Type": "AWS::EC2::Instance",
  "Metadata": {
     :
  },
  "Properties": {
    :
    "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
      "<script>\n",
      "cfn-init.exe ",
      "         --stack ", { "Ref" : "AWS::StackName" },
      "         --resource MyInstance ",
      "         --region ", { "Ref" : "AWS::Region" }, "\n",
```

```
      "cfn-signal.exe -e %ERRORLEVEL% ", {"Fn::Base64" : {"Ref" : "MyHandle"}}, "\n"
      "</script>\n"
    ]]}}
  }
},

"MyHandle" : {
  "Type" : "AWS::CloudFormation::WaitConditionHandle"
},

"MyWaitCondition" : {
  "Type" : "AWS::CloudFormation::WaitCondition",
  "DependsOn" : "MyInstance",
  "Properties" : {
    "Handle" : {"Ref" : "MyHandle"},
    "Timeout" : "600"
  }
}
```

For more details of using the `WaitCondition` resource, see the *AWS CloudFormation User Guide*.

Dealing with Stack Updates Using cfn-hup

With AWS CloudFormation you can update a stack after it has been created by using the `aws cloudformation update-stack` command or the `UpdateStack` API call. The stack update can be a simple change to a parameter value or a more complex update that updates, adds, or removes resources. AWS CloudFormation updates resource properties, adds new resources, or removes unwanted resources, but these changes might affect the applications running on instances in one of two ways:

1. Changing a resource in the template might require an update to the configuration of an instance. For example, if you add a database to the template for scaling, the application on an instance must be provided with the new connection string, and the instance might need to be restarted.
2. The metadata on the instance might have been updated. For example, you could update the version of a package that was deployed, add additional files, add additional packages, or run additional commands.

AWS CloudFormation provides the `cfn-hup` helper to reconfigure, restart, or update an application on an instance as part of the stack update process. The `cfn-hup` helper is a daemon that takes the actions that are specified in the resource metadata after it detects changes in resource metadata. You can use the daemon to make configuration updates on your running Amazon EC2 instances through `UpdateStack`.

For more information about `cfn-hup`, see cfn-hup in the *AWS CloudFormation User Guide*.

The `cfn-hup` helper must be configured to inspect the correct stack. This configuration is stored in the `cfn-hup` configuration file `cfn-hup.conf`.

Note: The `cfn-hup` helper uses the AWS credentials from the IAM role to retrieve the metadata. The IAM role is passed to the instance profile when the Amazon EC2 instance is created.

```
[main]
stack=<stack-name-or-id>
```

By default, every 10 minutes `cfn-hup` checks for changes in each resource path it is given. If a change to the requested metadata is detected, the user action is triggered. User actions (also known as *hooks*) are defined in a hook configuration file. Hooks are uniquely named, and each hook is configured in one section, as shown in the following example:

```
[hookname]
triggers=post.add|post.update|post.remove
path=Resources.<logicalResourceId>.Metadata<.optional.path>
action=<arbitrary shell command>
runas=<runas user>
```

This example includes the following elements:

- `hookname` (the section header) is a unique name for this hook.
- `triggers` is a comma-delimited list of conditions to detect. The conditions that are currently supported are `post.add`, `post.update`, and `post.remove.`
- `path` is the location of the metadata object. The `path` element supports an arbitrarily deep path within the `Metadata` block.
- `Action` is an arbitrary shell command (runs as specified).
- `runas` is an optional user. `cfn-hup` uses the `su` command to switch to the user.
- When the `action` is run, it runs in a copy of the environment that `cfn-hup` is in, with CFN_OLD_METADATA set to the previous value of `path` and CFN_NEW_METADATA set to the current value.

To support composition of several applications deploying change notification hooks, `cfn-hup` supports a directory `/hooks.d` that is located in the hooks configuration directory. Each file within this directory is parsed and loaded that uses the same layout as `hooks.conf`. If any hooks in `hooks.f` have the same name as a hook in `hooks.conf`, they are merged with `hooks.d`, overwriting any values in `hooks.conf` that both specify.

The hooks configurations are loaded when the `cfn-hup` daemon starts up, so new hooks require the daemon to be restarted. A cache of previous metadata values is stored at `/var/lib/cfn-hup/data/metadata_db` (not human readable). This cache can be blown away to force `cfn-hup` to run all `post.add` actions again.

Dynamic Software Updates Using `cfn-hup` and `cfn-init`

As described in the previous section, the `cfn-hup` helper is a small daemon that you can use to execute hooks when the metadata on a resource is changed through the AWS Management Console, the `aws cloudformation update-stack` command, or the `UpdateStack` API call. The

`cfn-init` function takes the packages and files that are defined in the metadata and installs them on your Amazon EC2 instance. By combining `cfn-hup` hooks with the `cfn-init` script, you can automatically install new versions of software when you change the metadata by updating the stack template.

The following example is a hook file that you might install by using the files section in the `AWS::CloudFormation::Init` metadata in your template:

```
"/etc/cfn/hooks.d/cfn-auto-reloader.conf" : {
  "content": {
    "[cfn-auto-reloader-hook]\n",
    "triggers=post.update\n",
    "path=Resources.WebServer.Metadata.AWS::CloudFormation::Init\n",
    "action=/opt/aws/bin/cfn-init "
          "  --stack ", { "Ref" : "AWS::StackName" },
          "  --resource WebServer ",
          "  --region     ", { "Ref" : "AWS::Region" }, "\n",
    "runas=root\n"
  }
}
```

In this file, we define a `cfn-hup` hook that looks for changes to the metadata that is defined in the `WebServer` resource in the stack and calls `cfn-init` if there is a change. If the metadata changes, `cfn-init` looks at all the versions of the packages that are defined for the `WebServer` resource and, if there has been a change, installs the version from the new template.

Because the templates are text files, you can version-control them just like other application artifacts. By doing so, you can version-control not only your AWS infrastructure configuration but also the set of packages installed on your instances.

If you specify a version of a package in the template, `cfn-init` attempts to install that version even if a newer version of the package is already installed on the instance. Some package managers support multiple versions while others do not. If you do not specify a version and a version of the package is already installed, `cfn-init` does not install a new version, it assumes that you want to keep and use the existing version.

Debugging Bootstrap and Update Scripts

You can see what events occur during the bootstrap sequence by viewing the following log files:

The `/var/log/cfn-init.log` log file contains the sequence of steps, along with any errors that resulted from running cfn-init.

The `/var/log/cloud-init.log` log file contains the output of all operations that were performed when running the CloudInit script. We recommend that, at least for debugging, you use the `–v` option on the bash command. This option shows what happened when the bootstrap script passed in information in from the user data field.

```
        "UserData"       : { "Fn::Base64" : { "Fn::Join" : ["", [
          "#!/bin/bash –v \n",
          "yum update -y aws-cfn-bootstrap\n",
          "/opt/aws/bin/cfn-init "
          "         --stack ", { "Ref" : "AWS::StackName" },
          "         --resource WebServer ",
```

```
          "         --region ", { "Ref" : "AWS::Region" }, "\n",
          "/opt/aws/bin/cfn-signal -e $? '", { "Ref" : "WaitHandle" }, "'\n"
        ]]}}
```

The `/var/log/cfn-hup.log` log file contains the output from running the `cfn-hup` helper and can be used to debug any issues when the `cfn-hup` daemon runs.

Security Considerations

To use the AWS CloudFormation bootstrap features, you need to provide AWS credentials to the bootstrap scripts. We strongly recommend that you assign an IAM role to on the EC2 instance when the instance is launched. By using an IAM role, no long-term secrets are defined in the template or stored in the metadata on the EC2 instance. When the IAM role is used, temporary security credentials are created and used to access AWS services such as AWS CloudFormation. These temporary credentials expire after a short time, making it harder to compromise the credentials and reducing the risk and exposure if the credentials are compromised.

A Complete Example

The following sample is the complete template that installs WordPress by using the AWS CloudFormation helper scripts and the metadata that is defined in the template. You can download the template from the following location: https://s3.amazonaws.com/cloudformation-templates-us-east-1/WordPress_Bootstrap.template

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",

  "Description" : "AWS CloudFormation Sample Template WordPress_CloudFormation: WordPress is web software you can
use to create a beautiful website or blog. This template installs a highly-available, scalable WordPress
deployment using a multi-az Amazon RDS database instance for storage. It demonstrates using the AWS CloudFormation
bootstrap scripts to deploy WordPress. **WARNING** This template creates an Amazon EC2 instance, an Elastic Load
Balancer and an Amazon RDS database instance. You will be billed for the AWS resources used if you create a stack
from this template.",

  "Parameters" : {

    "KeyName": {
      "Description" : "Name of an existing EC2 KeyPair to enable SSH access to the instances",
      "Type": "String",
      "MinLength": "1",
      "MaxLength": "255",
      "AllowedPattern" : "[\\x20-\\x7E]*",
      "ConstraintDescription" : "can contain only ASCII characters."
    },

    "InstanceType" : {
      "Description" : "WebServer EC2 instance type",
      "Type" : "String",
      "Default" : "m1.small",
      "AllowedValues" : [ "t1.micro", "t2.micro", "t2.small", "t2.medium", "m1.small", "m1.medium", "m1.large",
"m1.xlarge", "m2.xlarge", "m2.2xlarge", "m2.4xlarge", "m3.medium", "m3.large", "m3.xlarge", "m3.2xlarge",
"c1.medium", "c1.xlarge", "c3.large", "c3.xlarge", "c3.2xlarge", "c3.4xlarge", "c3.8xlarge", "g2.2xlarge",
"r3.large", "r3.xlarge", "r3.2xlarge", "r3.4xlarge", "r3.8xlarge", "i2.xlarge", "i2.2xlarge", "i2.4xlarge",
"i2.8xlarge", "hi1.4xlarge", "hs1.8xlarge", "cr1.8xlarge", "cc2.8xlarge", "cg1.4xlarge"],
      "ConstraintDescription" : "must be a valid EC2 instance type."
    },
```

```
    "SSHLocation": {
      "Description": " The IP address range that can be used to SSH to the EC2 instances",
      "Type": "String",
      "MinLength": "9",
      "MaxLength": "18",
      "Default": "0.0.0.0/0",
      "AllowedPattern": "(\\d{1,3})\\.(\\d{1,3})\\.(\\d{1,3})\\.(\\d{1,3})/(\\d{1,2})",
      "ConstraintDescription": "must be a valid IP CIDR range of the form x.x.x.x/x."
    },

    "DBClass" : {
      "Default" : "db.m1.small",
      "Description" : "Database instance class",
      "Type" : "String",
      "AllowedValues" : [ "db.t1.micro", "db.m1.small", "db.m1.medium", "db.m1.large", "db.m1.xlarge",
"db.m2.xlarge", "db.m2.2xlarge", "db.m2.4xlarge", "db.m3.medium", "db.m3.large", "db.m3.xlarge", "db.m3.2xlarge",
"db.r3.large", "db.r3.xlarge", "db.r3.2xlarge", "db.r3.4xlarge", "db.r3.8xlarge", "db.m2.xlarge", "db.m2.2xlarge",
"db.m2.4xlarge", "db.cr1.8xlarge"],
      "ConstraintDescription" : "must select a valid database instance type."
    },

    "DBName" : {
      "Default": "wordpressdb",
      "Description" : "The WordPress database name",
      "Type": "String",
      "MinLength": "1",
      "MaxLength": "64",
      "AllowedPattern" : "[a-zA-Z][a-zA-Z0-9]*",
      "ConstraintDescription" : "must begin with a letter and contain only alphanumeric characters."
    },

    "DBUsername" : {
      "Default": "admin",
      "NoEcho": "true",
      "Description" : "The WordPress database admin account username",
      "Type": "String",
      "MinLength": "1",
      "MaxLength": "16",
      "AllowedPattern" : "[a-zA-Z][a-zA-Z0-9]*",
      "ConstraintDescription" : "must begin with a letter and contain only alphanumeric characters."
    },

    "DBPassword" : {
      "Default": "password",
      "NoEcho": "true",
      "Description" : "The WordPress database admin account password",
      "Type": "String",
      "MinLength": "8",
      "MaxLength": "41",
      "AllowedPattern" : "[a-zA-Z0-9]*",
      "ConstraintDescription" : "must contain only alphanumeric characters."
    },

    "MultiAZDatabase": {
      "Default": "false",
      "Description" : "Create a multi-AZ MySQL Amazon RDS database instance",
      "Type": "String",
      "AllowedValues" : [ "true", "false" ],
      "ConstraintDescription" : "must be either true or false."
    },

    "WebServerCapacity": {
      "Default": "1",
      "Description" : "The initial number of WebServer instances",
```

```
      "Type": "Number",
      "MinValue": "1",
      "MaxValue": "5",
      "ConstraintDescription" : "must be between 1 and 5 EC2 instances."
    },

    "DBAllocatedStorage" : {
      "Default": "5",
      "Description" : "The size of the database (Gb)",
      "Type": "Number",
      "MinValue": "5",
      "MaxValue": "1024",
      "ConstraintDescription" : "must be between 5 and 1024Gb."
    }
  },

  "Mappings" : {
    "AWSInstanceType2Arch" : {
      "t1.micro"    : { "Arch" : "PV64"   },
      "t2.micro"    : { "Arch" : "HVM64"  },
      "t2.small"    : { "Arch" : "HVM64"  },
      "t2.medium"   : { "Arch" : "HVM64"  },
      "m1.small"    : { "Arch" : "PV64"   },
      "m1.medium"   : { "Arch" : "PV64"   },
      "m1.large"    : { "Arch" : "PV64"   },
      "m1.xlarge"   : { "Arch" : "PV64"   },
      "m2.xlarge"   : { "Arch" : "PV64"   },
      "m2.2xlarge"  : { "Arch" : "PV64"   },
      "m2.4xlarge"  : { "Arch" : "PV64"   },
      "m3.medium"   : { "Arch" : "PV64"   },
      "m3.large"    : { "Arch" : "PV64"   },
      "m3.xlarge"   : { "Arch" : "PV64"   },
      "m3.2xlarge"  : { "Arch" : "PV64"   },
      "c1.medium"   : { "Arch" : "PV64"   },
      "c1.xlarge"   : { "Arch" : "PV64"   },
      "c3.large"    : { "Arch" : "PV64"   },
      "c3.xlarge"   : { "Arch" : "PV64"   },
      "c3.2xlarge"  : { "Arch" : "PV64"   },
      "c3.4xlarge"  : { "Arch" : "PV64"   },
      "c3.8xlarge"  : { "Arch" : "PV64"   },
      "g2.2xlarge"  : { "Arch" : "HVMG2"  },
      "r3.large"    : { "Arch" : "HVM64"  },
      "r3.xlarge"   : { "Arch" : "HVM64"  },
      "r3.2xlarge"  : { "Arch" : "HVM64"  },
      "r3.4xlarge"  : { "Arch" : "HVM64"  },
      "r3.8xlarge"  : { "Arch" : "HVM64"  },
      "i2.xlarge"   : { "Arch" : "HVM64"  },
      "i2.2xlarge"  : { "Arch" : "HVM64"  },
      "i2.4xlarge"  : { "Arch" : "HVM64"  },
      "i2.8xlarge"  : { "Arch" : "HVM64"  },
      "hi1.4xlarge" : { "Arch" : "PV64"   },
      "hs1.8xlarge" : { "Arch" : "PV64"   },
      "cr1.8xlarge" : { "Arch" : "HVM64"  },
      "cc2.8xlarge" : { "Arch" : "HVM64"  },
      "cg1.4xlarge" : { "Arch" : "HVMGPU" }
    },

    "AWSRegionArch2AMI" : {
      "us-east-1"      : { "PV64" : "ami-7c807d14", "HVM64" : "ami-76817c1e", "HVMG2" : "ami-9c13ecf4",  "HVMGPU"
: "ami-c6867bae"  },
      "us-west-2"      : { "PV64" : "ami-1b3b462b", "HVM64" : "ami-d13845e1", "HVMG2" : "ami-6d8cf15d",  "HVMGPU"
: "NOT_SUPPORTED" },
      "us-west-1"      : { "PV64" : "ami-a8d3d4ed", "HVM64" : "ami-f0d3d4b5", "HVMG2" : "ami-84494fc1",  "HVMGPU"
: "NOT_SUPPORTED" },
      "eu-west-1"      : { "PV64" : "ami-672ce210", "HVM64" : "ami-892fe1fe", "HVMG2" : "ami-1138f166",  "HVMGPU"
```

```
: "ami-972fe1e0"  },
      "ap-southeast-1" : { "PV64" : "ami-56b7eb04", "HVM64" : "ami-a6b6eaf4", "HVMG2" : "ami-7a1a4528",  "HVMGPU"
: "NOT_SUPPORTED" },
      "ap-northeast-1" : { "PV64" : "ami-25dd9324", "HVM64" : "ami-29dc9228", "HVMG2" : "ami-4d3b734c",  "HVMGPU"
: "NOT_SUPPORTED" },
      "ap-southeast-2" : { "PV64" : "ami-6bf99c51", "HVM64" : "ami-d9fe9be3", "HVMG2" : "ami-010c683b",  "HVMGPU"
: "NOT_SUPPORTED" },
      "sa-east-1"      : { "PV64" : "ami-c7e649da", "HVM64" : "ami-c9e649d4", "HVMG2" : "NOT_SUPPORTED", "HVMGPU"
: "NOT_SUPPORTED" },
      "us-gov-west-1"  : { "PV64" : "ami-ab4a2d88", "HVM64" : "ami-a54a2d86", "HVMG2" : "NOT_SUPPORTED", "HVMGPU"
: "NOT_SUPPORTED" },
      "cn-north-1"     : { "PV64" : "ami-cab82af3", "HVM64" : "ami-ccb82af5", "HVMG2" : "NOT_SUPPORTED", "HVMGPU"
: "NOT_SUPPORTED" }
    }
  },

  "Resources" : {


    "ElasticLoadBalancer" : {
      "Type" : "AWS::ElasticLoadBalancing::LoadBalancer",
      "Metadata" : {
        "Comment1" : "Configure the Load Balancer with a simple health check and cookie-based stickiness",
        "Comment2" : "Use install path for healthcheck to avoid redirects - ELB healthcheck does not handle 302
return codes"
      },
      "Properties" : {
        "AvailabilityZones" : { "Fn::GetAZs" : "" },
        "LBCookieStickinessPolicy" : [ {
          "PolicyName" : "CookieBasedPolicy",
          "CookieExpirationPeriod" : "30"
        } ],
        "Listeners" : [ {
          "LoadBalancerPort" : "80",
          "InstancePort" : "80",
          "Protocol" : "HTTP",
          "PolicyNames" : [ "CookieBasedPolicy" ]
        } ],
        "HealthCheck" : {
          "Target" : "HTTP:80/wordpress/wp-admin/install.php",
          "HealthyThreshold" : "2",
          "UnhealthyThreshold" : "5",
          "Interval" : "10",
          "Timeout" : "5"
        }
      }
    },

    "WebServerSecurityGroup" : {
      "Type" : "AWS::EC2::SecurityGroup",
      "Properties" : {
        "GroupDescription" : "Enable HTTP access via port 80 locked down to the load balancer + SSH access",
        "SecurityGroupIngress" : [
          {"IpProtocol" : "tcp", "FromPort" : "80", "ToPort" : "80", "SourceSecurityGroupOwnerId" : {"Fn::GetAtt"
: ["ElasticLoadBalancer", "SourceSecurityGroup.OwnerAlias"]},"SourceSecurityGroupName" : {"Fn::GetAtt" :
["ElasticLoadBalancer", "SourceSecurityGroup.GroupName"]}},
          {"IpProtocol" : "tcp", "FromPort" : "22", "ToPort" : "22", "CidrIp" : { "Ref" : "SSHLocation"}}
        ]
      }
    },

    "WebServerGroup" : {
      "Type" : "AWS::AutoScaling::AutoScalingGroup",
      "Properties" : {
        "AvailabilityZones" : { "Fn::GetAZs" : "" },
        "LaunchConfigurationName" : { "Ref" : "LaunchConfig" },
        "MinSize" : "1",
        "MaxSize" : "5",
        "DesiredCapacity" : { "Ref" : "WebServerCapacity" },
```

```
    "LoadBalancerNames" : [ { "Ref" : "ElasticLoadBalancer" } ]
  }
},

"LaunchConfig": {
  "Type" : "AWS::AutoScaling::LaunchConfiguration",
  "Metadata" : {
    "AWS::CloudFormation::Init" : {
      "configSets" : {
        "wordpress_install" : ["install_cfn", "install_wordpress" ]
      },
      "install_cfn" : {
        "files": {
          "/etc/cfn/cfn-hup.conf": {
            "content": { "Fn::Join": [ "", [
              "[main]\n",
              "stack=", { "Ref": "AWS::StackId" }, "\n",
              "region=", { "Ref": "AWS::Region" }, "\n"
            ]]},
            "mode"  : "000400",
            "owner" : "root",
            "group" : "root"
          },
          "/etc/cfn/hooks.d/cfn-auto-reloader.conf": {
            "content": { "Fn::Join": [ "", [
              "[cfn-auto-reloader-hook]\n",
              "triggers=post.update\n",
              "path=Resources.LaunchConfig.Metadata.AWS::CloudFormation::Init\n",
              "action=/opt/aws/bin/cfn-init ",
                       "         --stack ", { "Ref" : "AWS::StackName" },
                       "         --resource LaunchConfig ",
                       "         --configsets wordpress_install ",
                       "         --region ", { "Ref" : "AWS::Region" }, "\n"
            ]]},
            "mode"  : "000400",
            "owner" : "root",
            "group" : "root"
          }
        },
        "services" : {
          "sysvinit" : {
            "cfn-hup" : {
              "enabled" : "true",
              "ensureRunning" : "true",
              "files" : ["/etc/cfn/cfn-hup.conf", "/etc/cfn/hooks.d/cfn-auto-reloader.conf"]
            }
          }
        }
      },

      "install_wordpress" : {
        "packages" : {
          "yum" : {
            "php"       : [],
            "php-mysql" : [],
            "mysql"     : [],
            "httpd"     : []
          }
        },
        "sources" : {
          "/var/www/html" : "http://wordpress.org/latest.tar.gz"
        },
        "files" : {
          "/tmp/create-wp-config" : {
            "content" : { "Fn::Join" : [ "", [
```

```
                  "#!/bin/bash\n",
                  "cp /var/www/html/wordpress/wp-config-sample.php /var/www/html/wordpress/wp-config.php\n",
                  "sed -i \"s/'database_name_here'/'",{ "Ref" : "DBName" }, "'/g\" wp-config.php\n",
                  "sed -i \"s/'username_here'/'",{ "Ref" : "DBUsername" }, "'/g\" wp-config.php\n",
                  "sed -i \"s/'password_here'/'",{ "Ref" : "DBPassword" }, "'/g\" wp-config.php\n",
                  "sed -i \"s/'localhost'/'",{ "Fn::GetAtt" : [ "DBInstance", "Endpoint.Address" ] }, "'/g\" wp-
config.php\n"
                ]]},
                "mode" : "000500",
                "owner" : "root",
                "group" : "root"
              }
            },
            "commands" : {
              "01_configure_wordpress" : {
                "command" : "/tmp/create-wp-config",
                "cwd" : "/var/www/html/wordpress"
              }
            },
            "services" : {
              "sysvinit" : {
                "httpd" : { "enabled" : "true", "ensureRunning" : "true" }
              }
            }
          }
        }
      },
      "Properties": {
        "ImageId" : { "Fn::FindInMap" : [ "AWSRegionArch2AMI", { "Ref" : "AWS::Region" },
                        { "Fn::FindInMap" : [ "AWSInstanceType2Arch", { "Ref" : "InstanceType" }, "Arch" ] } ]
},
        "InstanceType"   : { "Ref" : "InstanceType" },
        "SecurityGroups" : [ {"Ref" : "WebServerSecurityGroup"} ],
        "KeyName"        : { "Ref" : "KeyName" },
        "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
                       "#!/bin/bash -xe\n",
                       "yum update aws-cfn-bootstrap\n",

                       "/opt/aws/bin/cfn-init ",
                       "         --stack ", { "Ref" : "AWS::StackName" },
                       "         --resource LaunchConfig ",
                       "         --configsets wordpress_install ",
                       "         --region ", { "Ref" : "AWS::Region" }, "\n",

                       "/opt/aws/bin/cfn-signal -e $? '", { "Ref" : "WebServerWaitHandle" }, "'\n"
        ]]}}
      }
    },

    "WebServerWaitHandle" : {
      "Type" : "AWS::CloudFormation::WaitConditionHandle"
    },

    "WebServerWaitCondition" : {
      "Type" : "AWS::CloudFormation::WaitCondition",
      "DependsOn" : "WebServerGroup",
      "Properties" : {
        "Handle" : {"Ref" : "WebServerWaitHandle"},
        "Timeout" : "900"
      }
    },

    "DBInstance" : {
      "Type": "AWS::RDS::DBInstance",
      "Properties": {
```

```
        "DBName"            : { "Ref" : "DBName" },
        "Engine"            : "MySQL",
        "MultiAZ"           : { "Ref": "MultiAZDatabase" },
        "MasterUsername"    : { "Ref" : "DBUsername" },
        "DBInstanceClass"   : { "Ref" : "DBClass" },
        "DBSecurityGroups"  : [{ "Ref" : "DBSecurityGroup" }],
        "AllocatedStorage"  : { "Ref" : "DBAllocatedStorage" },
        "MasterUserPassword": { "Ref" : "DBPassword" }
      }
    },

    "DBSecurityGroup": {
      "Type": "AWS::RDS::DBSecurityGroup",
      "Properties": {
        "DBSecurityGroupIngress": { "EC2SecurityGroupName": { "Ref": "WebServerSecurityGroup"} },
        "GroupDescription"      : "Frontend Access"
      }
    }
  },

  "Outputs" : {
    "WebsiteURL" : {
      "Value" : { "Fn::Join" : ["", ["http://", { "Fn::GetAtt" : [ "ElasticLoadBalancer", "DNSName" ]},
"/wordpress" ]]},
      "Description" : "WordPress Website"
    }
  }
}
```

Getting the AWS CloudFormation Helper Scripts and Templates

The AWS CloudFormation helper scripts are available from the following locations:

- The Amazon Linux AMI has the AWS CloudFormation helper scripts installed by default in */opt/aws/bin.*
- The Windows AMIs provided by Amazon Web Services have the AWS CloudFormation helper scripts and the ability to execute scripts in user data installed by default.
- The AWS helper scripts are available in the Amazon Linux AMI yum repository (the package name is *aws-cfn-bootstrap*).
- In addition, the helpers are available in other formats. For more details go to: http://aws.amazon.com/developertools/AWS-CloudFormation/4026240853893296
- The templates that are used in the previous examples are available, along with many other sample templates, on the AWS CloudFormation sample template site at http://aws.amazon.com/cloudformation/aws-cloudformation-templates/