# Chapter 8. Building Reusable Components

It is the goal of any time-pressed systems administrator to avoid duplication of work where possible. There is no need to spend time building ten servers when you can build one and clone it, or implement a configuration management system that can configure ten servers as easily as one.

Within the context of AWS, there are many ways to work smarter instead of harder. Remember that AWS gives you the building blocks you need to build your infrastructure. Some of these blocks can be reused in interesting ways to remove tedious steps from your workflow.

As an example, consider an application that runs in three environments: development, staging, and production. Although the environments differ in some important ways, there will definitely be a lot of overlap in terms of the AWS resources required and the application’s configuration. Considering the reuse of resources will save a lot of time as your infrastructure grows, and will let you take advantage of the flexibility that makes cloud hosting so useful.

This chapter looks at some of the ways in which AWS components can be designed for optimal reusability in order to reduce development time and reduce time spent on operations and maintenance.

# Role-Based AMIs

It’s common in both AWS infrastructures and traditional datacenters to assign each instance or server a role that describes the functions it will perform. A web application consists of multiple roles: serving web requests, processing asynchronous tasks, providing a database, and so on.

The most popular configuration management tools provide some method of implementing a role-based architecture. In fact, it might be said that the raison d'être of configuration management tools is to provide a way to assign a role to a server or virtual instance. Applying a set of Puppet modules (or Chef recipes, or Ansible playbooks) to an instance prepares it to perform its role.

The speed with which cloud computing allows you to bring new instances online makes it even more feasible to adopt this approach fully and design your infrastructure so that each instance performs one role, and one role only. When it took days or weeks to bring a new server online, it was much more tempting to add “just one more” role to an already overburdened server.

This raises the question of how AMIs can be used to facilitate this approach. If an instance performs only a single role, do you need one AMI per role? Not necessarily. It is possible to create an AMI that can perform multiple roles, as we saw in [Chapter 5](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch05.html#example_application_stack).

When the instance is launched, the configuration management tool will be run. This launch-time configuration transitions the instance from the launch state to the configured state, at which point it should be ready to begin performing its role.

Using a single AMI for all roles means that each instance launched from this AMI will need to perform a lot of role-specific configuration at launch time. An AMI should always contain just the bare minimum number of installed software packages that are required by its instances. In this architecture, the AMI will need the packages required by each and every role. This leads to a lengthy launch-time configuration process, which will increase the amount of time it takes for an instance to be ready to perform its regular duties.

At the other end of the spectrum, creating an individual AMI for each role results in a much shorter launch-time configuration process, although at the cost of an increase in time spent creating and managing AMIs.

Making and testing AMIs is not difficult, but it is time-consuming. If you are using an automated AMI creation process, this cost becomes a lot easier to bear.

A third option uses a different approach to building a base AMI that can be reused for all roles. With this method, the base AMI contains all software packages required to perform any of the roles in your infrastructure. For example, it might include packages for a database server, web server, and in-memory caching server. However, none of the services are configured to start when the AMI is launched. Instead, the configuration management tool takes responsibility for starting services. This reduces the amount of time taken to perform the launch-time configuration, because software packages do not need to be downloaded and installed, only configured and started.

One downside of this third approach is that the base AMI might need to be rebuilt more frequently than role-specific AMIs. If your base AMI includes both PostgreSQL and Nginx, the release of an urgent update to either package will necessitate rebuilding the AMI and replacing any instances that are running the old version. Failing to do so would result in running insecure versions of core software, or result in running two versions of the base AMI, which will quickly become inconvenient to manage.

No matter which method you choose, you will need some way of mapping a running instance to a particular role, which is the topic of the next section.

## Mapping Instances to Roles

Once you know that each instance will perform a given role, how do you apply that role to the instance? The answer to this question comes in two parts: assigning roles in AWS, and making this information available to your configuration management system. The second task is covered in the next section.

Amazon offers two main ways to assign roles to instances or, looking at the problem from a higher level, pass arbitrary information to your instances: user data and tags. Of course, it is also possible to store such information in a database such as RDS, SimpleDB, or your own database.

User data is usually the easiest method for a number of reasons. Instances can access user data without needing any IAM access credentials. User data can be retrieved from a static URL, which makes it usable from almost any programming language.

The most basic method of using user data to control role assignation would be to use the entire User Data field to specify the role. This value would then be available to scripts running on the instance, for example:

#!/bin/bash

ROLE=$(curl http://169.254.169.254/latest/user-data)

echo My role is $ROLE

In practice, you might already be using user data for something else and not want to use the entire field for the role. In this case, you should move to a more suitable data structure, such as JSON or key/value pairs.

Using tags is slightly more complicated because they can be retrieved only from the AWS command-line tools or APIs, the latter requiring the use of a programming language with a suitable AWS client library. An IAM account must be used to access the APIs, which means that any use of tags requires access to a set of IAM access credentials.

The simplest way to provide these credentials is to use IAM roles so that credentials do not need to be hardcoded into scripts or retrieved from user data. Instead they can be automatically detected by the underlying client library used by your scripts (such as Boto, introduced in [“Launching from Your Own Programs and Scripts”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch02.html#first_steps_launching_from_programs)).

Tags have three advantages over user data. First, they are key=value pairs by design. This fits in neatly with the idea of mapping a role to an instance: we simply need to create a tag named role and give it a value such as web-server or db-server.

The second advantage of tags is that they can be queried from outside the instance far more easily than user data. Tags make it possible to perform an API query to retrieve a list of instances tagged with a role such as web-server. Performing the same query for user data would involve listing all your instances, parsing the user data, and compiling a list of matching instances.

The final advantage of tags is more of a business reason than a technical one. Tags can be used by Amazon’s billing system to produce an itemized list of instances divided into groups based on the arbitrary tags you have defined in your Cost Allocation Report. The same source of data informs both technical and business decisions.

[Example 8-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch08.html#example_using_ec2_tags) shows an example of retrieving tags from a Python script.

##### ***Example 8-1. Using EC2 tags***

from boto.utils import get\_instance\_metadata

metadata = get\_instance\_metadata()

my\_instance\_id = metadata['instance\_id']

conn = boto.ec2.connect\_to\_region("us-west-1")

reservations = conn.get\_all\_instances(filters={'instance-id': my\_instance\_id})

instances = [i for r in reservations for i in r.instances]

# instance-id is a unique identifier so it is safe to assume there is only one instance

instance = instances[0]

# Iterate through the tags, printing the keys and values

for key, value in instance.tags:

print "Key is %s. Value is %s" % (key, value)

This script does not include any IAM credentials. It assumes that the instance it is running on has been assigned an IAM role. Boto will automatically use IAM role credentials if they are available.

## Patterns for Configuration Management Tools

As mentioned earlier, reusability is a core goal for many configuration management tools, the entire purpose of which is to reduce duplication of effort by automating common tasks such as creating files or installing software packages. This section shows how this role-based design pattern can be used within Puppet, building on the information in the previous section. Note that, apart from the syntax used, there is nothing specific to Puppet about this pattern. It can be implemented in all the configuration management tools of which I am aware.

The usual modus operandi of Puppet is to use the hostname of an instance to control which configurations are applied to that instance. This is practically useless within AWS, because hostnames are automatically generated based on the IP address of the instance. It is possible to work around this by setting the hostname to a “useful” value before running Puppet—a valid tactic that is used by companies such as Pinterest. In our case, however, we want to bypass hostnames completely and use the role attribute that we assigned by way of user data or EC2 tags.

To find out information about the environment in which it is running, Puppet depends on a tool named Facter. Facter is responsible for providing Puppet with “facts” about the system it is running on. These facts can then be used within your Puppet modules. The instance’s hostname is a fact. Puppet modules make this available via the $HOSTNAME variable.

Facter has built-in support for EC2, which means that it will automatically provide certain EC2-specific data to Puppet. Facter will query all the available meta and user data variables and provide them as facts to Puppet. For example, the AMI ID is available at a URL with a structure like http://169.254.169.254/latest/meta-data/ami-id. Facter will automatically set the $ec2\_ami\_id fact to this value. Note that the variable name is prefixed with EC2\_, and any dashes are replaced with underscores.

For the sake of our example, let’s assume that we are using JSON-formatted user data with our instances, and our web server has been launched with the following user data:

{ "role": "web",

"environment": "dev" }

This JSON object declares two attributes: a role and an environment.

Facter populates the $ec2\_user\_data variable to make this information available immediately to Puppet. However, Facter does not know that this data is JSON formatted: as far as it is concerned, the user data is simply a string of text.

Puppet’s stdlib module provides the parsejson function required to extract the keys and values you need from the JSON object. The function converts a given text string into a JSON object and returns the result as a hash. Once the data is in a hash, you can access the role and environment attributes and use them in conditional statements within Puppet modules, as shown in [Example 8-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch08.html#example_user_data_roles_and_puppet).

##### ***Example 8-2. User data roles and Puppet***

node default {

require stdlib

$userdata = parsejson($ec2\_userdata)

$role = $userdata['role']

case $role {

'web': {

require role::www::dev

}

'db': {

require role::db::dev

}

default: { fail("Unrecognized role: ${role}") }

}

}

This example shows how the $role attribute can be used to control which modules are applied to each instance. The $userdata['environment'] variable could be used to provide a further level of abstraction, so that the live environment uses the role::www::live module, and the development environment uses role::www::dev.

For the sake of brevity, I have not included the www and db Puppet modules. These are simply Puppet modules that perform tasks such as installing Nginx or PostgreSQL.

###### NOTE

The module layout—in this case, role::www::dev and role::db::dev—is based on Craig Dunn’s [Design Puppet → Roles and Profiles](http://www.craigdunn.org/2012/05/239/) blog post. This is a great way to separate business logic (“What should this instance be doing?”) from technical details (“How should this instance be configured?”), and is particularly useful when adopting this pattern in AWS.

User data is only one way of providing information to AWS instances so that it can be made available to Puppet. The other option is to create tags on the instance and make these available to Puppet.

Unfortunately, tags support is not built into Facter as easily as user data. This is a minor hurdle to bypass, though—Facter makes it easy to add facts by way of a plug-in architecture. Plug-ins are simply Ruby scripts placed in a particular directory, where the name of the script is also the name of the fact that will be returned. Facter executes all the plug-ins while gathering facts about the system.

[Example 8-3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch08.html#example_ec2_tag_facts) shows an example Facter plug-in that retrieves all the tags for the instance and make them available to Puppet.

##### ***Example 8-3. EC2 tag facts***

require 'facter'

require 'json'

if Facter.value("ec2\_instance\_id") != nil

instance\_id = Facter.value("ec2\_instance\_id")

region = Facter.value("ec2\_placement\_availability\_zone")[0..-2]

cmd = <<eos

aws ec2 describe-tags

--filters \"name=resource-id,values=#{instance\_id}\"

--region #{region}

| jq '[.Tags[] | {key: .Key, value: .Value}]'

eos

tags = Facter::Util::Resolution.exec(cmd)

parsed\_tags = JSON.parse(tags)

parsed\_tags.each do |tag|

fact = "ec2\_tag\_#{tag["key"]}"

Facter.add(fact) { setcode { tag["value"] } }

end

end

For more information about adding custom facts to Facter, and to find out where on your system this plug-in should be located, see the [Custom Facts documentation page](http://docs.puppetlabs.com/guides/custom_facts.html).

With this plug-in in place, we can launch an instance and assign role and environment tags to it, instead of passing this information as user data. The code shown earlier in [Example 8-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch08.html#example_user_data_roles_and_puppet)has to be modified so that, instead of parsing JSON from the $ec2\_userdata variable, it obtains the same information by using the $ec2\_tag\_role and $ec2\_tag\_environment variables, for example:

node default {

case $ec2\_tag\_role {

'web': {

require role::www::dev

...

Although this section has focused on Puppet, the same result can be achieved with most other configuration management tools. The general principle of providing information to the instance at launch time, and then using this information later to control the instance configuration, can be used from configuration management tools or your own scripts.

# Modular CloudFormation Stacks

CloudFormation stacks can also be designed to make them more suitable for reuse in different parts of your application. This section presents one of the most popular methods of reaching this goal.

This method uses the AWS::CloudFormation::Stack resource, which lets you embed one CloudFormation template within another. That is, a parent CloudFormation stack can create a number of child stacks. The parent stack can provide input values to the child stack and access its output values. This means that the parent stack can create multiple child stacks and use the outputs of one stack as the parameters of another stack. In this design pattern, each child stack can be self-contained and highly focused on a particular task, such as creating EC2 instances or an RDS database.

###### NOTE

To use embedded templates, you need to provide a TemplateURL parameter, which tells CloudFormation where to download the stack template file. This file must be in an S3 bucket configured to serve its contents over HTTP. For more information, see the [AWS Stack Properties](http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-stack.html) documentation.

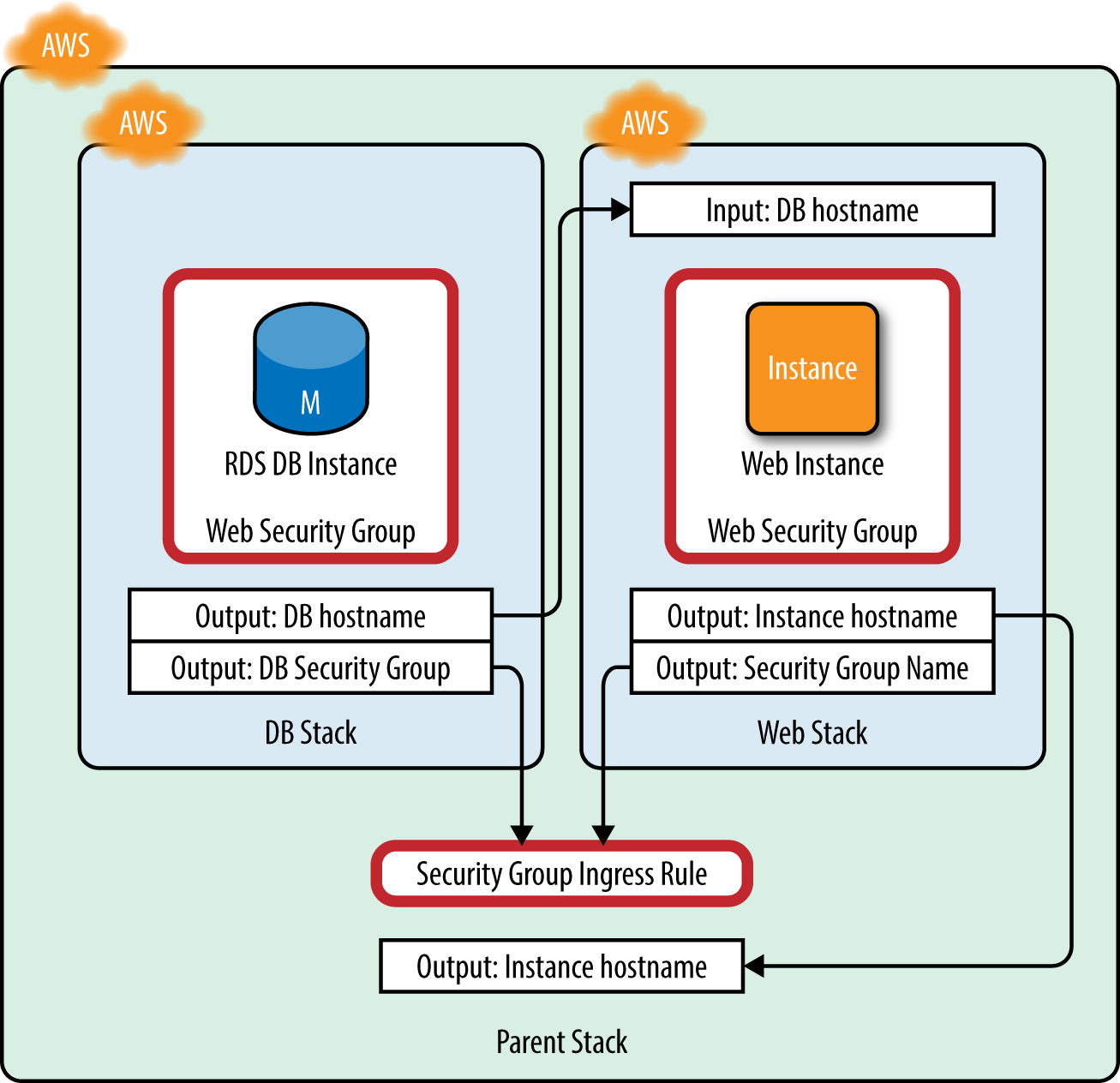
The parent stack is responsible for tying all these components together and providing the foundation your application needs to run. This is illustrated in [Figure 8-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch08.html#figure_cloudformation_parent_child_stacks).

The architecture in the figure consists of three CloudFormation stacks: the parent stack, the DB stack, and the web stack. The DB stack is responsible for creating an RDS instance and placing it in a security group. The web stack creates an EC2 instance, also in a security group.

When the RDS instance is created, it is assigned a unique hostname generated by Amazon, which cannot be predicted in advance. How then can you let the instance know the hostname of the database instance so that it knows where to send data requests? The answer comes in the form of parameters and outputs. These can be used to provide data when launching a stack and to retrieve dynamic stack attributes after it has been created.

In this case, the DB stack outputs the hostname of the RDS instance. The parent stack uses this value as an input when creating the web stack. In turn, the web stack passes this value to the instance as user data or a tag, so it can be used in your configuration management software. More information on using parameters and outputs with embedded stacks can be found on Amazon’s [Stack Resource Snippets](http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/quickref-stack.html) page.

CloudFormation attempts to automatically determine which order the resources should be created in, so that parameters and outputs can be passed around successfully. In this example, CloudFormation would detect that the output from the DB stack is used as an input to the web stack, and would therefore create the DB stack first.



###### *Figure 8-1. Embedded CloudFormation stacks*

###### TIP

If you need more control over the order in which CloudFormation resources are created, you can use the [DependsOn](http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-dependson.html) attribute to override CloudFormation’s automatic ordering.

Note that neither the Web nor DB stacks create any security group rules. Instead, these are created in the parent stack through the use of the AWS::EC2::SecurityGroupIngress resource type.

This resource allows you to create security group ingress rules and assign them to existing security groups. Why are these created in the parent stack? This is a personal preference more than a technical requirement. It would be possible, for example, to pass the DB security group ID as an input to the web stack and allow the web stack to create the necessary ingress rules, permitting the instance to access the database.

But the opposite is not possible: to pass the web security group as an input to the DB stack, you would need to create the web stack before the DB stack. That would make it impossible to provide the DB hostname as an input to the web stack, meaning the instance would not know the address of the database server.

By creating the ingress rules in the parent container, you are simplifying the DB and web stacks. They should not be concerned with creating security group rules, as it can be argued that ingress rules are not specifically tied to the function of the stack.

Moving some of the resource creation—such as ingress rules—to the parent stack increases the reusability of the child stacks. Consider the familiar example of development and production environments. Both will need web and DB stacks, but the development instance should be accessible only from a specific list of IP addresses, whereas the production environment will be accessible from the public Internet.

To enforce this distinction, you could create a separate CloudFormation stack for each environment, each of which embeds the web and DB stacks as children. The development and production stacks would be almost identical, except when it comes to creating the security group ingress rules. When you want to make changes to the DB stack, you have only a single template to update.

This provides a clean way of breaking your infrastructure into logical components that can be reused to create flexible environments without duplication of effort.

###### TIP

CloudFormation will automatically tag your resources with some information about the stack itself. If you configure Cost Allocation Reports to track these tags, you get an extremely high-level overview of where your money is going.

Other tricks can be used in CloudFormation templates to allow easy recycling of stacks. Let’s suppose that we need each developer on the team to have his own development environment, which should be accessible via SSH at a given hostname such as mryan.dev.example.com.

The development stack template could accept an EnvironmentName input parameter, which is used to create a Route 53 resource record mapping the desired hostname to the instance’s public DNS name. Each developer can create his own copy of the stack, using the same template, entering his username when the stack is launched.

Although the CloudFormation template language is not as flexible as a full programming or scripting language, it can be useful to think of your CloudFormation stacks in the same way that you think of your scripts, programs, or even Puppet modules. Each of these presents various methods that can be used to dramatically reduce the time it takes to add new features or deploy new resources.