LEARNING CHEF VIA AZURE

<https://docs.microsoft.com/en-us/azure/chef/>

Contents

[Planning for Chef 1](#_Toc515464075)

[Look at the Core Components of Chef Server 3](#_Toc515464076)

[Installing Chef 4](#_Toc515464077)

[Demo: Deploy Base Environment in Azure with PowerShell 4](#_Toc515464078)

[Demo: Deploy Base Environment in Azure with Azure CLI 5](#_Toc515464079)

[Install and Configure Chef Server on Ubuntu 6](#_Toc515464080)

[Install and Configure Chef Management Console 7](#_Toc515464081)

[Configuring the Chef Workstation 8](#_Toc515464082)

[Bootstrapping Chef Nodes 13](#_Toc515464083)

# Planning for Chef

The Problem of Scaling Server Management

Welcome to this Pluralsight course on Planning and Installing Chef. My name's James Bannan. By the end of this course, you're going to have a very thorough understanding on how to plan for and install Chef Server, which is an exceptionally powerful infrastructure as code-driven platform that allows you to deploy and manage your infrastructure regardless of how many instances, regardless of what it's running, and where it is. In our industry at the moment, infrastructure only ever increases, so you need a solid platform that's going to scale with your efforts. Chef is that platform. This is Module 1, Planning for Chef. Throughout this module, we're going to take a look at a number of key components. We're going to take a look at why traditional infrastructure management doesn't scale. I remember back when I started in IT, you would maybe administer 10, 15, 20 servers. Now it's more likely to be 200, 300. We're going to look at how infrastructure as code, which is the unpinning of what Chef is, is the way forwards. We're going to look at what it is exactly about Chef that allows us to do infrastructure as code and what that looks like. We're going to go through the process of signing up the hosted Chef, even though that's not what we're going to use throughout the lab. It's a very important thing and it's still worth seeing it in action. And then finally, we're going to look at how Chef works. We're going to look at the structure and the infrastructural components. Let's frame what we're going to learn as a traditional problem. We have a business, any kind of enterprise. They have a numbers of servers and they use traditional per-machine management. There's an administrator or a team of administrators and they log onto each server in turn to do whatever administrative work is necessary: Leveraging remote desktop, SSH, WinRM, and so on. Over time, the on-premises infrastructure expands and management overhead increases accordingly. Then at some point, management says, Hey, what about this whole cloud thing? Let's go there. So, infrastructure is spun up, but we're still using the same administrative processes, we're still logging onto every server individually. So what occurs, our traditional per-machine management paradigm doesn't scale, we're wasting time, and the business is losing agility. If the business says, Hey, we want to implement something new, there's all this administrative overhead that has to come along for the ride and we don't want that. So, the answer is infrastructure as code. We define what we want our infrastructure to look like, how we want it to behave, and we encapsulate that into code. Once it's in code, all of a sudden, we get access to all of the benefits of code, specifically it's versionable. So, I can take a look at code and I can say, Okay, this is version 1.1. Awesome. I particularly need to deploy version 1.3, so I'll use that version instead. It's testable. Once it's encapsulated into code, you can leverage standard development tools to go through and automatically test that code to make sure that it's good before you deploy it. And very importantly, it is repeatable. That means because it's code, because it's encapsulated as code, it doesn't matter who runs it on what platform - well, hopefully - every time it gets run, it is the same result. And the platform that this particular business with all of these management infrastructure problems has chosen to use is Chef.

Examine a Sample Chef Recipe

The most basic unit of currency in Chef is a recipe. A recipe is a small or potentially very large amount of code that tells the infrastructural platform what to do, specifically what you want it to look like rather than how to get there. Recipes are bundled into cookbooks, cookbooks are deployed against your infrastructure, and the Chef agent processes them accordingly. Let's take a look at a sample Chef recipe. In this instance, we want Chef to work with a particular type of resource, in this case, a package. It could be a file, it could be a folder, it could be a service, it could be any number of resources available. The particular package that we want this recipe to agree to is httpd or Apache.

So this recipe would be used to set up an Apache web server. Now that we've specified the resource type and the actual resource that we want to work with, we need to specify an action. We want something to happen to this package. And the specific action that we want to occur is that we want it to install it. Note that in this particular recipe we're not telling Chef how to achieve a particular outcome. We're not telling it, First install this and then check for this and do this and so on. We're simply telling it what we want the end result to be. And the end result is that Apache gets installed. That's what we want. This is an example of idempotence. We don't care what the underlying platform is: Ubuntu, Red Hat. We don't care how Apache is installed, as long as it is. This is the sort of code we want to be working with. We provide the code that states our desired outcome, but does not state how to get there. And this is one of the absolute foundational principles of Chef. When planning for Chef, there are two main options: Hosted Chef or Chef Server. Hosted Chef is provided by Chef itself at chef.io, and we will go through that process momentarily. There are a number of things to look at when choosing between Hosted Chef or Chef Server.

Hosted Chef is fully posted, so its software is a service. Chef Server must be installed either in your own environment or in an infrastructure as a service provider like Azure, AWS or Rackspace. Hosted Chef is very, very easy to set up, whereas Chef Server has a more complicated installation process. Hosted Chef has no ongoing maintenance. chef.io looks after all of that behind the scenes and it's something that you never need to worry about.

Chef Server is a server within your own environments and, therefore, it needs to be maintained as any other server would be. Hosted Chef gives you very few options for customization. It is a service that Chef provides and if it meets all of your needs, excellent. If it does not, there's not much you can do about it. Whereas, Chef Server allows for very, very fine- grained control over how it is installed, where the infrastructure sits, and the various components contained within it. And finally, because Hosted Chef is a software as a service platform, you have very little control over the sorts of access speeds your servers are going to have. Whereas, access to Chef Server in your own environment gives you full control. Therefore, you can execute some extremely rapid deployments and configuration changes.

Sign up for Hosted Chef

So, let's go ahead and sign up for Hosted Chef. Even though we're not going to use it throughout the course, it's a very worthwhile process to go through, firstly to see just how easy it is, but also to see what we're going to expect when we set up Chef Server within our own environment. Fire up our web browser and head on over to chef.io, click on Get Chef, and then start the signup process for Hosted Chef. Enter a full name, a valid email address, and make up a username, and enter in a password, as well as your company. The company doesn't actually need to be the company that you work for. It can be a fake one. Agree to the terms of service and you're done. The first thing that Chef needs is an organization. An organization is something internal to the Chef Server itself. Now, we don't have an organization, so we are prompted to make a new one. This organization needs to have a unique, short name. The reason for that is that because Hosted Chef is publically accessible, all of the organization's short names make up externally accessible URLs. And as such, the one that you nominate must be unique to you. The signup process will tell you whether or not a name is unique or not. Once our organization has been created, we're automatically logged into the Chef console under the administration pane. Under Administration, we can access and modify certain things like other organizations, users, groups, and permissions. The Policy pane is where recipes get uploaded into cookbooks and cookbooks are made available to our Chef notes, as well as other resources like data bags, which are dumps of information like users and groups, which we wish to apply to our Chef nodes and environments. For example, production or dev test. And finally, under Nodes, we can see all of the servers that have registered against our Hosted Chef instance. Now at the moment, we don't have any. When we install Chef Server in our own environment, we will also see an empty pane as well. But as we Bootstrap nodes with the Chef agent, we will see them pop up here. And as we assign cookbooks to our Chef Nodes, as they run them, as they action them, that information is going to get rolled up into the Reports pane.

## Look at the Core Components of Chef Server

Now that we've had a look at Chef, let's take a deeper look at how Chef actually works.

* Chef Server is the core of Chef regardless of whether it's Hosted Chef or an on-premises installation like the one we're going to do.
* The Chef Development Kit sits on one or more administrative workstations. This is where you do your recipe creation, bundle those recipes into cookbooks, generate policies, and upload that to the Chef Server. It's also where you do your test-driven infrastructure where you build up your code and you test it to make sure that it's good before deploying it out into the live environment.
* Chef Server talks to and from Chef Client Nodes. These nodes can be running any number of managed platforms like Ubuntu or Windows. And these nodes can be either on-premises or in a cloud-hosted provider. As long as they can talk directly with the Chef Server, they can be managed. The Management console is an optional installation. When you install Chef Server, you don't actually get the Management console installed by default.

With Hosted Chef, the Management console is automatically there. But when we do an on-premises installation, the Management console is something we need to explicitly install after installing Chef Server. The reason for this is that as you will see, most of the work is actually done from the Administrative workstation via the Chef DK or Chef Development Kit. You could, in theory, do all of your work from Chef DK without ever having to log on to a Management console.

There are premium features of Chef Server, such as high availability and replicated federation. In especially large environments where a single Chef Server is not going to be enough to manage all of the Chef Client Nodes, high availability is going to be a very critical feature where we break out a number of Chef Server components like web server interfaces and we make that highly available across multiple nodes. Replicated federation ensures that information that we upload into Chef Server is available across all of our managed Chef Server instances. And there's an optional analytics platform that takes all of the data that is being rolled up from the Chef Client Nodes and makes that available for us to work with using Chef Server. Now let's take a look at the various infrastructural components that make up Chef Server. At the core, we have Chef Server. Chef Client is an application which is installed or bootstrapped onto a managed node or at least a node that we want to manage. Those nodes then talk through a public load balancer powered by NGINX. Users also talk directly to that load balancer as well, providing a single point of entry into the Chef Server environment. The management UI or the Chef Management Console is also accessed through the same API model and that talks directly to Chef Server. The management console is a Ruby on Rails 3.0 application and it manages and hosts all of the web management functionality for Chef. Knife is a component of the Chef DK, which is normally installed on an administrative workstation. The workstation can be running Windows, Linux, Unix or MAC OS X. It talks through NGINX and then into the Chef Server. Cookbooks, which uploaded into Chef Server, are written into Bookshelf, accessed by the Chef Server, and then made available to Chef Client Nodes. All information about recipes, cookbooks, and nodes is stored in a searchable index. Chef Server leverages RabbitMQ to access that index. And finally, PostgreS powers the database back into Chef Server. And so, we have reached the end of Module 1. What have we learned? Well, we've learned that traditional server management does not scale when you are talking hundreds or potentially thousands of server instances. Infrastructure as a code or where we define where we want the end states of our infrastructure to be allows us to scale very easily. And Chef is an exceptionally powerful platform that lets us declare what we want that end state to be and to get that end state applied. We know the core of how Chef Server works and we know what are the core components of Chef Server itself. So, let's move onto Module 2 where we are going to install Chef Server in a virtual machine running in Windows Azure.

## Installing Chef

Module Overview and Prerequisites

So, let's get started on Module 2: Installing Chef. In this module, we're going to take an administrative workstation. Now, this can be a machine that you already use or it can be a virtual machine that you provisioned especially for this course. We're going to

* install Azure PowerShell and the Azure command line interface. Depending upon the operating system that you are running is going to impact which one of these you can install. On Windows, you can install both PowerShell and CLI. If you're running Linux or Mac, you can only install Azure CLI.
  + Azure Powershell module: <https://docs.microsoft.com/en-us/powershell/azure/install-azurerm-ps?view=azurermps-6.0.0>
    - <https://stackoverflow.com/questions/38171877/the-term-add-azureaccount-is-not-recognized>
* From our administrative workstation and using either PowerShell or Azure CLI, we're going to talk to Windows Azure and we're going to generate a resource deployment consisting of networking, storage, and a virtual machine, which we'll be running Ubuntu Server 14.04 LTS. On that server, we're then going to install the main components of Chef Server, i.e. Chef Server itself, the management console, a reporting stack, we're going to make an admin user, we're going to create an organization and then assign that admin user to the organization so that we can log on.
* There are a couple of prerequisites, which I'm going to assume you have taken care of on your administrative workstation.
  + The first thing that you're going to need is a trial *account of Microsoft Azure.* In order to sign up for a trial account, you need a Microsoft Live account. So for example, user@outlook.com. If you already have access to Microsoft Azure - say you're in MVP - you can use that instead. *If your administrative workstation is running Windows, you'll need to install Azure PowerShell.* You do this by downloading and installing Microsoft Web Platform Installer. And there's the URL with installation instructions. The example I'm going to use throughout this course of a non-Microsoft administrative workstation is Ubuntu. And for this, we're going to need to install Azure command line interface. I've done the installation of Azure CLI by first installing nodejs-legacy and NPM. There are other mechanisms that you can use to get CLI onto your Ubuntu workstation or even onto your Mac if that's what you want to use. Throughout this course, you can use either Linux or Mac if you don't want to use Windows. There's nothing about this course which is particular to Windows.
  + So let's start off the environments. We're going *to generate an Azure Active Directory admin account. This isn't necessary if you're running Azure PowerShell on Windows because you can log on using your Microsoft accounts*. However, if you're running Mac or Linux, you can't authenticate to Azure using a Microsoft account. Therefore, we need an administrative account within AD. We'll then take a very quick look at the deployment resources. And finally, we will deploy our Chef environments. This is the base environments running a virtual machine, which we can then customize with Chef Server.

### Demo: Deploy Base Environment in Azure with PowerShell

So, we're logged onto our Windows machine.

1. The first thing that we're going to do is fire up PowerShell and authenticate to our Azure account. In this case, we're going to get prompted for a username and password. We're going to be using our live account. So for example, user@outlook.com. Behind the scenes, we'll be authenticated and then PowerShell now has access to our Azure tenet. If we do get azure subscription, we can see more information about our Azure tenet, specifically the name, ID, and so on. In our case, the name is a free trial because that's what we've signed up for. If we do a quick listening of the course resources for our Windows workstation, we can see we've got a couple of files.
2. We have a PowerShell script called Create Course Environment ARM - ARM in this case is Azure Resource Manager - and we've also got azuredeploy.json or JSON. This contains all of the information about the resources that we want to provision in Azure, so networking, public IPs, storage, virtual machines, and so on. If we're going to run the PowerShell scripts, we're going to get an error. The reason for this is that we are using the default execution policy of PowerShell scripts, which is to block any PowerShell script which is not signed. For the purposes of our lab environment, we will fix this by changing the execution policy to bypass, not something that you would necessarily want to do in production. Now we can run the scripts again and it's going to launch without an error. The script is going to read the information contained within the JSON file about where we want our resource group to be created, what name to give it and so on. So it's going to create a resource group called chip-lab and it's going to place it in westus and assign it to our free trial. One bit of information which it needs which it doesn't have is the admin password, so we'll enter that. Once done, it's going to verify that the JSON template is valid and then it's going to go through and create each of the resources which we need in order to provision our Chef environment. For the purposes of the course, I've sped the process up. You may find it'll take a few minutes longer.
3. And once done, our Azure resource deployment is complete and ready for us to start working with it. To see what that actually looks like at the back end, what we'll do is fire up a web browser and we'll log onto portal.azure.com, in this case using our Microsoft Live account. If we go to Browse All and take a look at all of our resources, we can see that all of the various resources that we specified in the JSON like public IP, virtual network, and specifically our Chef Server running Ubuntu have all been provisioned and are ready for us to start working with.

### Demo: Deploy Base Environment in Azure with Azure CLI

If you're going to deploy the course environment using Mac or Linux, then you're going to need Azure command line interface. You can also install Azure command line interface on Windows using Web PI. Personally, I prefer to use PowerShell when I'm using Windows. But as you will see, Azure CLI does offer you a really good administrative experience. So in this part of the module, I'm logged onto Ubuntu and that's where I'll do all of my work from. Using Azure CLI is a little bit different than Windows PowerShell in terms of the way we authenticate to Azure. Azure CLI doesn't give us the ability to authenticate using our Microsoft Live account.

So, we need to log onto Azure using our Microsoft Live account and then create a new administrative user in Azure AD. This is the user that we will use within Azure CLI to authenticate to Azure and build up our Chef environment. So the first thing we do is fire up our browser and log onto portal.azure.com using our Microsoft Live account. Once we've logged in, go to Browse All and then select Active Directory. This is going to launch us into the older management platform. The Microsoft portal or portal.azure.com is a repository of all of the new features which are coming to Azure, whereas manage.windowsazure.com is the older management portal. At the moment, AD can only be managed from the older portal.

Once it loads, go into the default directory, select users, and add user to generate a new user. In this case, I'm going to call my new user CLI so I know that that's my user for command line interface work. Give it a first name, a last name, and a display name, and change the role to global admin. Because of this, you will need to specify an alternate email address. This doesn't need to be a real one, but if you're planning on using this account long-term, the alternate email address should be a valid one. Get the temporary password and copy it to the clipboard because we're going to need that pretty soon.

Next, go down to Settings, click on Administrators, and we need to add a new admin for our Azure subscription. Type in the email address of the user that you've just created, click OK, and we're done. Now because this user is using a temporary password, we need to change it. So, sign out and then sign back in using that new administrative password into the temporary password and then nominate a new one. Sign in and the user can see the resources currently allocated to it, which means it is properly set up as an administrator in the Azure environment.

Now we can actually start doing some work. Now back onto the Linux machine. Type in Azure and if you've installed Azure CLI properly, you will receive a screen that looks something like this. What we want to do initially is we want to login. So, we'll type in Azure login and we'll be prompted for a username and password. We've authenticated and now we can access our Azure free trial. As with Windows, you will need to copy the course materials to your machine in order to complete this part of the course. The materials look very similar to the ones we used on Windows except the scripts are Shell scripts rather than PowerShell scripts. We've got the CreateCourseEnvironment using ARM scripts, we've got the azuredeploy.json, we've also got an azuredeploy.parameters.json and we'll take a look at that in more depth very shortly, and we've also got a GenerateNames shell script. This is the one we're going to use first. The reason for this is that when we create a new Azure deployment, we need to specify names for particular resources including storage accounts and a name for the public IP. Because these are publicly accessible, they have to be unique. If we simply made the name Chef Storage or Chef-Lab, then one person could do the course and everyone else would encounter problems. What the scripts are designed to do is to generate a random string and then append that to the name that we're going to use. In the Windows PowerShell scripts, that was automatically done in the one script. For the Shell scripts, I had to break it out over a couple of scripts. This has less to do with Linux functionality and says a lot more about my ability to code BASH scripts. So, if anyone has a few tips, please let me know. So as you can see, we've got a couple of new names, we've got a storage account, and we've got a DNS name for public IP. These have to be placed in the azuredeploy.parameters.json file. This file gets passed to azuredeploy.json, which reads it and then inputs those parameter values into the Azure resource deployment. First off, let's take the storage account name, open up azureparameters.json - in this case, I'm using Atom - and we'll replace the value in the file. We'll go through and do the same for the DNS name for the public IP. Note that the admin username and the admin password are already here. You can change these if you need to. And finally, everything is ready to go, so we're going to run the CreateCourseEnvironment ARM Shell script and in a relatively short space of time, we have a deployment of an operational and all of our resources have been created. Now, obviously I've sped this up for the purpose of the lab. It should take anywhere between about 5-10 minutes when you perform the same operation. And finally, we're going to log into portal.azure.com, refresh all resources, and there they all are: Public IP, virtual network, and most importantly, our Chef Server all ready to go.

### Install and Configure Chef Server on Ubuntu

Now that we've built our lab environments, we can go ahead and start installing Chef Server. All of the work done in this module from here on in is going to be done on the Chef Server itself. So, it doesn't actually matter whether you are doing it from Windows, off Linux or Mac. The process is exactly the same. The first step is to go and download the latest version of Chef Server from chef.io. The build updates fairly frequently, so it's worth jumping onto chef.io first and making sure that you're downloading the right version. In this case, I'll use wget to launch the download from the Chef Server itself, but you could also download the package to your workstation and then upload it to the virtual machine using SCP. Because we're running on Ubuntu, we're going to use the dpkg installer to install the DB file. And finally, once it's been installed, we'll run Chef Server control reconfigure, which will set up the Chef Server for initial use. So, let's jump into the download. We have to download Chef Server to the Azure virtual machine, install and configure Chef Server, and then we are ready to go. Before we get started, it's worth pointing out that chef.io have actually made a free pre-configured Chef Server image available in Microsoft Azure. In the future, this means that if you're looking to provision a Chef Server in Azure very rapidly, this is the preferred method. However, for the sake of this course, we're going to go through each it step-by-step because installing a preconfigured image hides many of the underlying, infrastructural tasks from you. In order to connect to our virtual machine to start the Chef Server installation, we first need to work out the external, fully-qualified domain name, which has been assigned to the public IP address, which has been assigned to our Chef lab resource group. We can use PowerShell to do this and there's an equivalent command for Azure CLI as well. Once we've got the external name, copy it to the clipboard and then open up your SSH tool of choice. In this case, on my Windows machine, I'm using PuTTY, but you can use anything that supports SSH. Because I've never connected to the virtual machine previously, I'm prompted to trust the thumbprint of the self-signed certificate that's being used to authenticate the SSH connection. Once done, we're prompted to login. The login name is Chef and the password is chefpass. The format of it is the same as in the Linux Azure Parameters JSON file. Our next step now is to download the latest Chef Server core package from chef.io. We're going to save it locally at /temp. Once done, we're going to trigger the installation of the downloaded package. It's important to note that all of the commands that you're typing in require elevation. Even though you're logged in with root level access, the installation process will complain if you don't elevate. So make sure that every command is prefixed with sudo. Now that the download is done, we'll trigger the installation. The installation package gets unpacked and then Chef Server core is installed and provisionally configured. At this point, we don't actually have a fully-functional Chef Server. That comes next. But if we type in Chef Server control, we can see that a provisional help menu is launched, showing us that Chef Server is actually installed. In order to make it functional, we need to type in Chef Server control reconfigure. And at this point, Chef Server runs through quite a lengthy process to initially provision the Chef Server. I've sped the process up quite considerably. In production, you'll probably find that this takes anywhere up to 10 minutes to complete. As the process is running through, make note of all of the infrastructural components, which we covered earlier on in Module 1. Components like knife, cookbooks, bookshelf, PostgreS, NGINX, and RabbitMQ are all being set up for the first time now. The reconfigure process sets up every aspect of the Chef Server, including permissions, symbolic links, and certs. It's also worth noticing that there are many references to something called Opscode. Opscode was the original company name for Chef. So, whenever you see a reference to Opscode, it is actually referencing Chef. Also, throughout the reconfigure process, make a note that Chef Server is calling multiple cookbooks internally in order to set itself up. These cookbooks are based upon recipes and recipes and cookbooks are something that we touched upon in Module 1. Later on in the course, we'll be building our own. You may've also noticed that unlike when we signed up for Hosted Chef, when we triggered the installation of On-Premises Chef, we weren't prompted for an organization name, short name or a username and password. So once our Chef Server is up and operational, we won't actually have anything to work with. So, that's going to be our next step once Chef Server is installed. Once complete, all the services will restart and our Chef Server is up and operational and ready for us to start customizing. Now we can create our first Chef administrative user, create an organization, and assign that new administrative user to the organization. This process is going to involve creating some certificates, so let's go ahead and create a new folder where we can store them. To create a new user, we're going to use chef-server-ctl user-create. Don't forget to use the sudo function first. We're going to specify a username, first name, last name, and email address, which doesn't need to be real, password, which is unfortunately in plain text, and then we'll specify a path where to store the user certificate. Once that's done, we can go ahead and create our new organization using chef-server-ctl or -create. We need to specify a short name. Unlike Hosted Chef, it doesn't need to be a unique name. We'll then provide a long name and we'll associate our administrative user and then provide the path where we can store the organization certificate. Now we're done and we can carry on with the installation and configuration of Chef Server.

### Install and Configure Chef Management Console

Now that we have a functioning Chef Server, an administrative user, and an organization, we can start to install some of the extra components. In this section, we'll go ahead and install and configure the Chef Management Console. We'll do that by telling Chef Server to do the installation by using Chef Server control. The component name is called Opscode-manage. Remember that Opscode is another word for Chef. We'll then run Chef Server control reconfigure and Opscode manage control reconfigure to fully install and configure the web-based management console. So, let's jump into it. We're going to install the management console, we're going to reconfigure Chef Server, configure the management console, and then we're going to connect to the management console externally using our new administrative user. So, over in our Chef Server, we'll type in sudo chef-server-ctl install Opscode-manage. This will trigger the Chef Server to start the management console installation. It's going to get done by leveraging the add-ons wrapper recipe within the private-chef cookbook. Again, for the purposes of the lab, I've sped up the process considerably. When I ran it, it took approximately 15 minutes. Once it's installed, we'll tell the Chef Server to do a reconfiguration run. Unlike the first time, this run will take much less time because Chef Serer is already up and operational. Once that's complete, we can now go and tell Opscode to reconfigure itself using Opscode manage control. Now that we've got the management console installed, we should be able to access it from the external, fully-qualified domain name of our public IP. So again, we'll use PowerShell to get that value and then copy and paste it into a browser. We should be presented with a warning that there's a problem with the website security certificate. This is expected. Behind the scenes, the management console is using a self-signed cert based upon the host name of the virtual machine. The host name is not the same as the external name, so there is a name mismatch and our browser is just warning us of this. We'll continue anyway. Sign in using the username and password of the administrative user that we created earlier. We should be able to sign in and if we go to Administration, we can see we've got our newly-created organization. And under Members, our administrative user has been assigned to this organization, so we are looking good.

Install and Configure Chef Reporting

The last Chef Server component that we're going to install is Chef Reporting. This is part of the Management Console and like the Management Console, we're going to install it by using chef-server-ctl. In this case, install Opscode-reporting. Like the Management Console, we will then trigger a reconfiguration run for both chef-server-ctl and the newly-installed opscode-reporting-ctl. So, let's jump into it. We're going to install reporting, reconfigure the Chef Server, and configure the reporting feature. We will also jump into the management console to verify that the reporting feature has been installed properly. Before we do the installation of Chef Reporting, let's jump back into the management console to see what we should expect once reporting is up and operational. In the console, if we click on Reports and then Run History, we should get an error, 404 "Not Found." This is because the management console is trying to pull content from the underlying reporting feature, which isn't there. So, let's cut back across to the Chef Server and type in sudo chef-server-ctl install opscode-reporting. The process for installing Opscode Reporting looks very similar to Opscode Manage and it takes a while, too. About 6-7 minutes in production. Once it's complete, we'll use sudo chef-server-ctl reconfigure and just like the management console run, this reconfigure should be done quite quickly. And finally, one more reconfigure run, this time using opscode-reporting-ctl and we are finished. Now, let's cut back across into the Management console. This time, when we go to reports and go to Run History, this time instead of an error 404, we actually have a rendered page. There is no information on this page at the moment because we have no managed nodes which are rolling data up into the Chef Server. If there were, we would see it here. So, congratulations. We've reached the end of a relatively lengthy and technically tricky module. Let's just do a quick overview of what we've achieved so far. So, we built our lab environment in Azure using Azure Resource Manager, either leveraging Azure PowerShell or Azure CLI. We then installed Chef Server on Ubuntu and we've installed and configured the Management Console and Chef Reporting. This sets the infrastructural foundation for the rest of the course where we can start working with cookbooks, recipes, our administrative workstation, and we can Bootstrap Chef Client onto Node again within Azure. So, well done on making it this far and let's get stuck into some more Chef.

### Configuring the Chef Workstation

Configuring the Chef Workstation

So let's get stuck into Module 3: Configuring the Chef Workstation. In this module, we're going to build up a workstation where we're going to do all of our Chef work. That workstation is then going to be connected with the Chef Server, which we built in Module 2. The first thing that we do on our administrative workstation is we're going to install Git. Because Chef's infrastructure is code, it's really important that you get familiar with the idea of source control. A lot of Windows administrators don't necessarily use this at the moment mostly because there hasn't really been much of a call for it. But in the Chef world, especially if you're a Windows admin, this is something that's going to be critically important. Once we've done that, we're then going to install the Chef Development Kit or Chef DK, which is how I will refer to it throughout the rest of the course. Chef DK comes with a number of components all bundled into one package. The fundamental of these components that we're going to be interested in is Chef Client, Knife, Ohai, Chef Zero. There are other components which are more critical when we start leveraging infrastructure as code, such as Berkshelf, Test Kitchen, ChefSpec, and Food Critic. Once complete, we're then going to customize our management shell - in this case, PowerShell - to ensure that we have all of the necessary management resources available to us. We're then going to download the starter kit from the Chef Server, which we built in Module 2 and we're going to initialize that as a Git repository. We're going to download all the necessary management certs, which we need to connect to our Chef environment. We're then going to generate a basic cookbook, a couple of recipes, and we're going to upload that to Chef Server. So, let's get stuck into the practical side. We're going to install Git, download and install ChefDK, configure the shell environment, and work with that shell environment to talk back to Chef Server. In order to get our Windows workstation set up as an administrative Chef workstation, the first thing that we're going to do is head on over to GitHub and download and install their desktop application: GitHub for Windows. The reason for this is because by default, Windows doesn't come with any source control tools installed. There are, of course, a number of agents and applications used for source control that we could install on our Windows workstation, but GitHub for Windows provides everything in the same package, so that's the one we're going to make use of. Likewise, there are loads of platforms that we could use for source control other than GitHub. Personally, in addition to GitHub, I also use GitLab because their free account also allows you to provision private repositories, which you can only do on GitHub using a paid account. While it's downloading it and installing, let's just take a moment to talk about why source control is so important when we're dealing with infrastructure as code. Because we're dealing with code, it's extremely important to be able to track changes as they occur throughout the life cycle of that code. It can be hard enough if you're only working by yourself to keep track of changes. Certainly when I've been working on PowerShell projects in the past, I've found myself making PowerShell scripts Version 1, PowerShell script Version 2, PowerShell script Version 3, and so on. It would've been a much better idea to have one script under version control, allowing me to keep track of changes as I make them. Certainly, that's what I do now and because of that, I no longer have the problem of wondering which file contains which change. It's all documented for me. If you're working in a team where there are a number of people making changes to different code snippets all the time, it becomes even more important to control how that code changes over time so that you can deal with issues when they arise of multiple people editing the same code at the same time and coming up with slightly different ideas as to how things should work. Now that we have Git available on our workstation, we can turn our attention to the ChefDK. This is available at chef.io as a discreet, downloadable package. Select the package for the platform that you want to install and trigger off the download. On Windows, don't get confused by the architecture type. There's only one available to you, which is x86 or 32-bit. This will install just fine on a 64-bit platform. In fact, in this environment, I'm running 64-bit Windows 10 and the installation will proceed just fine. Chef have also made the package available as an MSI, so if you're working in an Enterprise and you would like to distribute that package out using some kind of Enterprise mechanism like Microsoft Configuration Manager, Then MSI will allow you to do that extremely easily. Once complete, we now have everything in place, so we can move on and start configuring our management shell. First, launch ChefDK from the desktop icon. This will launch a custom PowerShell session. If we do a get-module, we can see that a custom Chef module has been imported. And then if we do a get-command on that module, we can see that we have a number of commandlets specific to Chef available to us. Let's compare that with a Git shell, which was installed for us when we installed GitHub for Windows. If we do a get-module, we don't have the Chef module, but we do have another one called posh-git or PowerShell-git. If we do a get-command on this module, we can see that we have a number of commandlets available to us, which have specific Git functionality. These are not available to us currently in the ChefDK environment, which is obviously not what we want because we'll be doing almost all of our work within the ChefDK shell and we don't want to have to jump between shells. So the first thing to do is to find out where the posh-git module is installed from. So if we do a get-module and then look at the module base, this will provide us with a file path. If we do a listing of that file path, we can see that the module file that we want is posh-git.psm1. PSM1 is a PowerShell module file. We want to add this module to our PowerShell profile, which will make it available to us every time we launch PowerShell. So if we take a look at the profile variable, we can see where the profile is meant to be. It doesn't currently exist there, so we'll do a new-item and we will create it. Then launch Notepad so that we can edit it. Using import-module, we'll make sure that the PowerShell git-module is automatically loaded whenever a PowerShell session is opened. But, we need to do a little bit more because there is some extra command line functionality, which we're going to want to access. From our Git shell, if we take a look at our environment path variables, we can see that there are a couple of entries specific to GitHub. These locations contain extra command line tools like Git. We're going to want to be able to use these. So, we'll take a copy of those paths and then go back into our PowerShell profile. Add those paths in and we'll use a command +=, which will add them to the environment every time PowerShell is opened. Close PowerShell down and now let's go back and open up the ChefDK. This time, when we do a get-module, we can see that we have access to both Chef, as well as posh-git. And then, just because we can, we'll go back in and we'll add set location to C:Pluralsight so that every time we open up PowerShell, it's always going to default to that location. It just makes life a little bit more convenient for us.

Configure ChefDK

Now that our administrative workstation is all set up and ready to go, we can start the customization process, which will allow us to connect to our Chef environment. The first thing that we're going to do is to download the starter kit from our Chef Server. A starter kit contains everything we need to connect an administrative workstation to a Chef Server, specifically PM certs and a knife configuration file, which tells our administrative workstation which URLs to use, which names, and so on. We're going to download the SSL certs from our Chef Server to ensure that ChefDK can communicate securely with it. And then finally, we're going to run through a verification process. The verification process involves telling Chef to make sure that all of the individual components installed locally are running the way they should be. And there's also an SSL check where we can make 100% sure that coms are working the way they ought to be. So, let's get stuck in. The first thing that we're going to do is to download the starter kit. As its name implies, a starter kit is a small downloadable package, which allows you to pre-configure the ChefDK and makes it easier to connect to an organization for the first time. You download the starter kit by going to the Chef Management Console, heading into Administration, and then choosing the organization which contains the starter kit that you want to download. If you had multiple organizations within your Chef Management Console, each of them would have a starter kit that was unique to them. The starter kit downloads as a zipped file, so the first thing that we need to do is to extract it to a local path. If we now go and take a look at that path, we can see that we have a new folder called chef-repo. The fact that it has repo in the name is a major hint that Chef considers this to be the foundation of a repository. In other words, that you'll be putting this folder under source control so that you can effectively manage any changes. If we go into that folder and take a look, you can see that we've got a preexisting structure. We have a .chef folder, which contains the PEM certs which we created when we provisioned Chef Server for the first time, the user cert, and the organization cert. We have a Cookbooks folder and as the name implies, this is where cookbooks and recipes are managed from and we also have a Roles folder. Roles are used to apply multiple cookbooks or multiple recipes to a discreet collection of Chef managed nodes. And we have a git-ignore file and a readme.md. These are two critical components of any Git repository. At the moment, our chef-repo is not a repository, so we're going to use git init in order to initialize it as a repository. The work that we did earlier by placing the GitHub file paths into our profile means that the Git command is going to work. If we had not have done that, we couldn't use the Git command from PowerShell except in the Git shell. Once done, we'll do a listing of the folder, except we'll only specify hidden attributes and there's our new .git folder. Now that we've got the base structure, let's open it up in a graphical editor so that we can see what we are working with. In this case, I'm using Atom, which is a cross-platform editor available from atom.io. We'll navigate to the chef-repo folder so we can open it up in Atom and explore it. So you can see in the .chef folder we've got our certs, but we also have a file called knife.rb. This is our knife configuration file. Knife, as you remember, is a core component of the ChefDK and it's an application that we're going to leverage a lot in order to do our work with Chef Server. The knife.rb file tells knife where the Chef Server is based upon the URL, what certs to use, and where those certs are located. If you think back to when we installed and accessed the Management Console, you'll recall that we had a cert error. This is because of a domain name mismatch. The Chef Server URL contained within the knife.rb file is the actual host name of the Chef Server. From the context of Azure, this is an internal name; it's not externally accessible and it's not the name that we are using in order to access the Management Console. This is where the error with a domain name mismatch comes from. Up to now, we haven't had to do anything about this because while a browser will present us with an error, it does give us the opportunity to skip past that error manually or a manual override. The ChefDK does not give us that opportunity. Part of the process of configuring the ChefDK is to download a trusted cert from Chef Server. If there is a domain name mismatch, then the ChefDK or knife won't be able to validate the SSL connection properly and the attempt will fail. Normally in production, one of the critical tasks is to ensure that the Chef Server is using a name which is internally resolvable. If the server has to be made externally accessible, then a proper cert needs to be issued to the server. But in our lab environment, we can use a work around so that we don't need to do this. We're going to find out the public IP address, which has been assigned to our Chef environment. We will then edit the administrative workstation host file and assign that public IP address to the internal name of the Chef Server. This will allow us to use that name even though we are external. We're now going to run knife and we're going to tell it to jump out and fetch the SSL cert, which has been assigned to Chef Server. Note that it's reading the knife.rb file, so it's attempting the download using the internal URL of the Chef Server. Because we have edited the host file and added the external public IP, the connection attempt works. Otherwise, it would time out and fail. It's important to note that in order to communicate securely with a Chef Server, the ChefDK must have direct access to the certs that have been allocated to that server. ChefDK and knife does not make use of the local cert store on a Windows box or the equivalent on Mac or Linux, for example, Keychain. So even if you have provisioned a Chef Server in your own environment and assigned it with a certificate signed by a CA, which your administrative workstation already trusts, this is not enough for knife to be able to securely talk with Chef Server. We need to go down the knife SSL fetch path no matter what. If we jump back into Atom, we can see that now we have a new folder, Trusted Certs, and we have the Chef Server cert. This is the cert that knife is going to use when it communicates with our Chef Server in Azure. In order to make sure that everything is good, we'll run knife SSL check and knife will communicate with our Chef Server and validate that the SSL cert is good. And the last thing we're going to do is run Chef verify. This is going to go through and run an automated check across every component of our ChefDK. It checks for any broken dependencies and makes sure that everything is in proper working order. The verification of our ChefDK should pass without incident. So, now we have full verification, full secure communication, functionality, so it's time to start writing some Chef.

Start Using ChefDK

Now that ChefDK is installed and configured, we can start making use of it. The first thing we're going to do is a basic exploration of ChefDK, looking at the Chef repo and the starter cookbook, which is contained within it. We will then make a couple of basic Chef cookbooks for Linux and for Windows with associated recipes. And then using knife, we'll upload all of that to Chef Server. The first thing that we'll do is go in and open Atom where our Chef repository is already open. Expand our cookbooks and we'll take a look at the starter cookbook, which is part of the starter kit. As with all Chef cookbooks, there's a metadata file. This file contains identifying information about the cookbook, such as a name and a description. Under recipes, there is a default file. This is the Chef recipe which is run by default when this cookbook is called. You can have as many recipes as you like, but every cookbook has a default recipe. This recipe tells Chef to write to the log saying, "Welcome to Chef" based upon the node and a starter name. The node and the starter name are Chef variables. Node is a dynamic Chef variable based upon the name of the managed Chef node, whereas the starter name is a user-defined variable. If we go up into attributes, there is a default file here as well and we can see that the default value for starter name is Sam Doe. So when this particular cookbook gets run, it's going to log Welcome to Chef, node name Sam Doe. If we go into Roles, we can see that there is a starter role. Unlike recipes, there is not a default role and node is either a member of one or more roles or it isn't at all. If we take a look at the starter role, we can see that it has a run list allocated to it. In this case, it's going to run a recipe and the recipe is the starter cookbook. This means that we're in a server which is allocated to this role, processes the allocated run list, it's going to run the default recipe in the starter cookbook. If we wanted to run a different recipe, we would need to specify it here. However, you never need to specify the default recipe. Simply referencing the cookbook will automatically run the default recipe. And there's also an override statement where the starter name variable has been changed to my name. So this is going to overrule the entry in the Attributes file. Now we can go ahead and start creating our own cookbook. Navigate into the cookbooks folder and do a listing and you can see that we've only got the one to work with. Using knife, we're going to type in knife cookbook create and call it lab-linux. Knife is then going to build a properly formatted cookbook called lab-linux in the Cookbooks folder of our Chef repository. Do another listing and you can see it's been created properly. Jump into the folder and we have all the necessary folder structure that we need in order to work with a cookbook, so attributes, providers, recipes, resources, templates, and so on. It's important to note we also have a readme.md file and a changelog.md file, both of which are very important for source control. And of course, we have a metadata file. So, let's cut across into Atom and take a look at our new cookbook. It's automatically refreshed in our fog list, so if we open up the recipes, we can see we've got a default recipe. There's absolutely nothing in it at the moment, so this cookbook has no functionality. And if we take a look at the metadata file, we can see it looks pretty much the same as the metadata file in our starter kit. So, let's go ahead and add our own recipe to the cookbook. We can right-click in recipes and select New File. We'll get prompted for a file name and we'll call this apache.rb. Whenever you come across a .rb file within Chef, it is actually talking about Ruby. So ultimately, quite a lot of the work that you will do with Chef will involve writing Ruby code. Fortunately, there are loads of examples that you can use so you don't have to learn all of the syntax from scratch and also Atom has quite a good IntelliSense when it comes to using Ruby, which is a big help if you're just learning. Let's open up our newly-created recipe and we'll just put in a description. So, this is going to install the Apache web server. If you think back to Module 1 and the example recipe that we took a look at there, this is the recipe we're going to use here. So, we're going to specify a package, - in this case, it's httpd - we're going to provide a do function because we want Chef to perform some kind of action, and then we're going to tell it what that action actually is, which in this case is install. Note that the IntelliSense has come to our aid and it's automatically done the correct formatting and it's provided the end statement. Now, let's go down and look at the roles in our starter kit. We've got the starter role, but let's make a new one called lab-linux and we will place our newly-created recipe as part of the run list for this new role. Go to the starter role and we can shamelessly copy and paste code straight out of that into our new role and we'll just modify the values accordingly. So, we only want the name, the description, and the run list. So, the name is lab-linux, we'll give it a description of "Pluralsight role for Linux Server," and we'll change the run list from recipe starter to "recipe lab Linux Apache." Now, note that we have to specify the recipe here. If we didn't, it would just attempt to run the default recipe, which as we saw, has nothing in it. So, we need to tell it which recipe we're going to use. Now, we're going to go ahead and create a new cookbook, this time for use with Windows rather than Linux. The process is very similar except that in order to work effectively with Windows, we're going to need some extra dependencies. Traditionally, Chef has been for working with Linux and Unix. Support for Windows has come along a little bit later, but it is very strong and it's improving all the time. Chef and the Chef community maintain a large repository of pre-built cookbooks, which are under constant development. The one that we're going to make use of is the Windows cookbook. Yes, it's an original name, but is an official Chef cookbook. There are a number of ways that we can add preexisting Chef cookbooks into our ChefDK environment. We can use knife or as in this case, we're going to clone the repository straight from GitHub. This is not the production-ready repository, but rather the development repository. For our purposes, that's going to be just fine. So, we'll clone it down from GitHub to our local workstation and if we do a listing of our Cookbooks folder, we can see we now have a brand new cookbook called Windows. Use the knife function to generate a new cookbook called lab-windows, verify that we now have a new cookbook, and once that's done, we can jump back into Atom and verify that it's ready for us to start working with. We're going to use this cookbook to tell Chef to enable the Telnet client feature on our manage Windows servers. We're going to use the default recipe for this one. There's no particular reason why we couldn't spin up a new recipe, but it's just more for example purposes. So again, we're putting a description, which is to enable the Windows Telnet client feature, and now we're going to use a slightly different provider. In the Apache recipe for the lab-linux cookbook, we use the package provider. This is something that Chef natively understands. It does not natively understand the windows\_feature package provider. However, that has been defined in the Windows cookbook and that's why we downloaded it. So, we'll type in windows\_feature, we'll specify Telnet-client as the name, and the rest of the syntax is exactly the same as the Apache recipe. In order to tell Chef that there is a dependency upon the Windows cookbook, we need to go down and edit the metadata file for our lab-windows cookbook, at the end adding depends Windows. Any cookbook can have any number of dependencies on other cookbooks. What we're telling Chef here is that in order to function correctly, the Windows cookbook must be available to the Chef client. And now that we have a Windows recipe with all of the necessary dependencies in place, we'll go through and create a new role that we can assign this cookbook to. The process is exactly the same as previously: Select new file, give it a name - in this case, we're going to call it lab-windows - and shamelessly copy and paste everything from lab-linux into lab-windows and simply change all the values. So, we're calling it lab-windows, Pluralsight lab role for Windows Servers, and the recipe that we need to specify is lab-windows. Remember, we don't need to specify the recipe name because we're using the default recipe. Now that we have created cookbooks, recipes, and roles, we want to upload them all to Chef Server so that any managed node can potentially make use of them. So the first thing we'll do is from our ChefDK, we'll type in knife cookbook list. This is going to connect through to Chef Server and get a listing of all the cookbooks that are currently present. The list is going to come back empty. So, let's upload the ones that we have locally. We'll type in, knife cookbook upload --all. You could also do -a. This is going to attempt the upload process, but we're going to be presented with an error. The specific error is that cookbook Windows has a dependency which is missing. The dependency is chef\_handler and the dependent version is any version equal to or above Version 0, so any version at all. Now, if we'd read the readme for the Windows cookbook, we would've seen that chef\_handler is a dependency, but it's good to know that knife, rather than just allowing us to upload cookbooks for which there is a broken dependency is actually running a check and informing us. Like the Windows cookbook, the chef\_handler cookbook is actively developed by Chef. So, we can download that from GitHub just like we did for Windows. Once it's downloaded, we'll attempt the upload again and this time, it's going to pass without error. If we tell knife now to jump out to Chef Server and list all the available cookbooks, now we have a list. We want to do the same thing for the roles, so let's go into our Roles folder and just do a quick listing to make sure that all the necessary files are there. We'll tell knife to talk to Chef Server and get a list of currently installed roles. And as we can see, there are none. The command to upload them is slightly different. We're not telling knife to upload a file, but rather we're telling knife to inform Chef Server to create a role from a file and then we're telling it where that file which contains the necessary role definition is. So we'll create the lab-linux role and the lab-windows role. And now if we do a listing, we now have roles available to us. So, let's switch over to the Management Console and see if we can see any changes. If we go into the Policy tab, we can see that we now have a list of available cookbooks: chef\_handler, lab-linux, lab-windows, and so on. And each of them has a version allocated to it. This is the version number, which has been pulled out of the metadata file. If we go into roles, we can see that we now have lab-linux and lab-windows available to us. And if we select lab-windows, we can see that there is a run list, which has been allocated to the role with a specific version. And that brings us to the end of Module 3. Congratulations. We've done quite a lot of work throughout this module. Let's just do a very quick recap. We installed the ChefDK on our administrative workstation along with Git. We also configured the management shell so that our ChefDK PowerShell window has access to all the tools that we need to work effectively. We downloaded the starter kit from our Chef Server and used that to generate our first Chef repository. We used knife to connect to Chef Server, validating that our knife configuration file was all correct and we downloaded SSL certs and verified that they were good, too. We then made a couple of cookbooks, we made recipes and roles, and we used the ChefDK, specifically knife, to upload all of that into Chef Server. So, let's move onto Module 4 where we will provision some extra instances in Azure and have them connect through to our Chef Server.

### Bootstrapping Chef Nodes

#### Deploy Client Servers in Chef-lab Azure Resource Group

Welcome to Module 4 and the final module in this course, Bootstrapping Chef Nodes. All the work that we've done throughout the previous three modules is all going to come together in this module. We're going to build upon our chef-lab environment, which we provisioned in Azure and we're going to extend it by accommodating two new virtual machines. One of them running Windows Server and the other one running Ubuntu Server. From our administrative workstation, we're going to use knife bootstrap to install Chef client on both of these virtual machines and make sure that they talk and register properly with Chef Server. We're then going to take the cookbooks and the roles, which we built earlier and apply them to our managed nodes. We're going to instigate a Chef client run and make sure that our managed servers have been updated and modified according to those recipes. So the first thing we need to do is to use Azure Resource Management, which is the methodology that we use to provision the Chef environment in the first place to install our new Chef Servers. First, we'll install Ubuntu, then Windows, and we'll verify that they can both talk to Chef Server on our internal Azure virtual network. Make sure you grab the lab resources because you will need them in order to provision our client servers. There are four files we will need: Two PowerShell scripts and two JSON files. The PowerShell scripts each reference a JSON template, which is the description of the infrastructure that we want to provision. The first virtual machine we want to build is the Linux agent. The process looks pretty much the same as it did for the Chef Server. We're using the same version of Ubuntu, which is 14.04 LTS. The difference in this case is that we can reuse existing resources like storage accounts and the virtual network. We will be provisioning a new public IP address so that we can hit this particular virtual machine independently of all the other IPs. This isn't necessarily something we would want to do in production, but for the sake of a lab, it's perfectly fine. As with the Chef Server, we will need to specify an admin password and the rest of the build should be relatively fast. And once it's complete, we now have a new Ubuntu Server up and operational. Making use of Azure Resource Manager to provision our infrastructure is another terrific example of using infrastructure as code. The definition for every infrastructural component, such as the virtual network, the virtual machines, network interfaces, public IPs, and so on is all templated as code. This means that the ideas that we're looking to promote, leveraging Chef as our management platform - again, infrastructure as code - fit really well with our base underlying infrastructure. Using Azure PowerShell to query our resources, we can see that we now have two virtual machines instead of one. So, we've got Chef Server and Linux Client. Now we can start provisioning our Windows Client. Again, the process is exactly the same as for our Linux Server. PowerShell queries a JSON template, which provides all the necessary information that Azure needs to jump out and provision the virtual machine. As with the Linux Server, we're going to reuse the virtual network and the storage account, which we've already provisioned. Azure Resource Manager is often referred to as Version 2 as Azure's infrastructure as a service. If you've been working with Azure infrastructure as a service up to this point in the traditional management console, you've been working with Azure Service Management or Version 1. Azure Service Management allows for some very rich automation and scripting, but Version 2 is where the incredible power of templated infrastructure comes into play. Once you've completed this course, I recommend that you take a look at some of the other terrific Pluralsight courses on Azure Resource Manager. Once our Windows Server has been successfully provisioned, again, jump out using Azure PowerShell to see what resources we have and we've now got three virtual machines in our chef-lab environment. Now that our Windows and Ubuntu Servers have been provisioned, the next thing that we want to verify is that they can both communicate with our Chef Server on the Azure virtual network, which we provisioned earlier on in the course. When we provisioned our two servers, we specified that an external public IP address also be provisioned and then assigned to each server. The reason for this is so that we can remotely connect from our administrative workstation. But the external public IP addresses are not the method that each virtual machine are going to use to communicate with each other. They're going to do that internally. But, in order for that to work we must verify that every server can talk to all the others. Using Windows PowerShell, we will extract the information from each public IP address, which has been provisioned in our chef-lab environment and get the external name. The first machine that we're going to check is our Ubuntu box. The process of SSH-ing into our Ubuntu Server is exactly the same as it was for our Chef Server. We'll get the external public IP and the associated name, put that into PuTTY, and SSH into the box. And then we'll run a ping test on the internal name of the Chef Server. This is the same name which is in the knife configuration file that's part of our chef-repo. The ping test works and more importantly, it's pinging on 10.0.0.4, which is the internal IP address, which has been assigned to the Chef Server on our Azure virtual network. Next, we want to do exactly the same thing with our Windows box. So we will use Azure PowerShell to find out the external name and this time instead of SSH, we will RDP into the box. Once we're in, fire up our command prompt and run the same ping test. Again, the ping test works, we chose that both of our agents have full internal connectivity to our Chef Server using the host name, which the Chef Server is registered with. Now, we can move onto bootstrapping the Chef agent from our administrative workstation onto each virtual machine.

#### Bootstrap Client Servers Using Knife

Now that we've extended our chef-lab environment with two extra virtual machines, we can proceed to install or to bootstrap the Chef agent onto both of them. First, we'll go through and bootstrap Ubuntu, then we'll bootstrap Windows. The bootstrapping process for Windows is a bit different to that for Ubuntu, so we will spend a little bit of extra time there. And then we will verify that the agents installed can communicate properly and are registering properly with the Chef Server. The first thing we'll do is use knife to get a list of the nodes currently registered with our Chef Server using knife node list. You have to be in the chef-repo folder for this to work properly. If the knife command can't find the configuration file, it'll just complain. Because we have no nodes registered against our Chef Server, the list is going to come back empty. So, the first machine that we want to bootstrap is our Ubuntu Server. We'll use Azure PowerShell to query our Chef Lab environment and get the fully-qualified domain name of the public IP address assigned to the Linux Server. Once we've got that, we'll use knife to go and bootstrap the Chef agent onto our Linux Server. The command is fairly long. Knife boot strap is the foundation. We then specify the name of the server which we use in order to communicate with it, bearing in mind that this is not the internal name of the server, but that doesn't matter in this case. This is just the name which the administrative workstation makes use of. Knife has its own SSH functionality and that's what it's going to use to talk to the Ubuntu box. It's going to SSH into it and do all the work locally on the server itself. So, apart from a couple of small local text files, no information, no large packets are uploaded from the administrative workstation. All the work is done locally on the remote server itself. We need to specify our SSH user and the SSH password. It is, unfortunately, in plain text. If you don't want to specify a username and password, you can use certificate-based authentication. We then want to specify a node name. This is the name which the remote server, our Ubuntu Server, is going to use to register itself against the Chef Server. The bootstrapping process requires elevated rights on the remote server. So we specify sudo and we want to see every aspect of the process, so we specify verbose. The bootstrapper tells the remote server to jump out to opscode.com and to download an installation Shell script. This script is a small package, which determines the operating system which the server is running and then redirects it to download the correct version of the Chef agent. In this case, it's downloading for Ubuntu, which is what we are running. The package is downloaded to the temp folder on the remote server, unpacked, and installed. And once it's done, the bootstrapper informs the Chef agent to perform its first run against the nominated Chef Server. If we had specified a run list as part of our bootstrapping process, which we can do, then this would get picked up during the very first run. As we can see, this server has an empty run list, so nothing is going to run. We can also see that it's using the node name, which we specified right at the start. Once that's done, we'll use knife node list again to jump out the Chef Server and see if we have any registered nodes and we do. Our Ubuntu Server is happily registered and talking to Chef Server. So now we can move onto bootstrapping and Windows Server. As already mentioned, Chef is built upon Ruby and so the ChefDK comes with its own embedded Ruby engine. A lot of the functionality of the ChefDK is provided by Ruby Gems, which are self-contained Ruby packages. From a command line, if you type in "gem list," you'll receive a list of all of the locally installed Gem packages. The gem that we need for knife to be able to work with Windows is knife-windows. And we can see that we have a local copy of that already installed. This is the version of knife-windows which came with the ChefDK when we downloaded and installed it. However, this version of knife-windows is a bit out of date. So in order to work with the latest and greatest versions of Windows, we're going to want to update that. We can update it directly from the command line, but if you want to find out more about what gems are available and what versions existing gems are already up to, you can jump over to rubygems.org and browse the library there. So, type in "gem update knife-windows" and knife will reach out to rubygems.org and update to the latest version, which is 1.0.0 at the time of recording. Notice, though, that we received a warning saying that a particular path was not available within our environment path list. This is where the gems are installed, so we're going to need to very quickly add that to our environment path using PowerShell and then we'll check that path again to make sure that the change has been made. Now we're ready for knife to work with Windows. Using knife to bootstrap the Chef agent onto a Windows machine generally involved the use of WinRM or Windows Remote Management. This is a different management and communications protocol to SSH. By default, Windows does not have the ability for an administrative user to connect remotely using SSH. So rather than install SSH on every Windows machine in our environment, we can make use of the underlying management stack, which is WinRM. But because our administrative workstation is also running Windows, the connection attempt will fail. We need to configure WinRM at both sides of the communications path. We do this by launching an administrative window and typing in WinRM quick config, accept the defaults, and basically that's it. We are done. It's that fast. We need to do the same on the remote Windows Server, which we want to bootstrap as well. Finally, because our administrative workstation cannot by default communicate securely with our remote server because they do not share a single source of secure identity like AD, we have to add the name of the remote server that we're going to use into the WinRM TrustedHosts list. Once that's complete, our administrative workstation will be able to securely open a WinRM connection to our remote Windows Server using credentials which we specify. The first thing we need to do is use Azure PowerShell to get the fully-qualified domain name of the public IP address which has been assigned to our Windows machine. Once we've got that, we'll then specify the knife bootstrap command, which will be used to connect to the remote machine using WinRM and initiate the download and install of the Chef agent. The format of the command is very similar to the one that we used for Ubuntu. Knife bootstrap is the foundation, but because we're working with a different platform than Ubuntu, we need to specify that we're using Windows. Even though we are using WinRM, there are actually multiple ways that you can connect to a Windows machine. However, in this case, we're using WinRM, so we specify WinRM. We then specify the fully-qualified domain name that we retrieved earlier and instead of specifying an SSH user, we specify a WinRM user and an associated password. Yes, again, unfortunately it is in plain text. And we also want to specify a node name, which is the name that the Windows machine is going to use to register itself against the Chef Server. Knife waits for the remote server to respond before initiating the bootstrap process. Once it does respond, knife renders a batch file locally on the remote server. This is the file that's going to be used to execute the bootstrap process. The Windows Server then leverages PowerShell to initiate a wget PowerShell script, which will jump out to chef.io and initiate the download of the latest MSI package, which is the installer package used on Windows. Once complete, the installation is initiated on the remote server in the same way that the DEB package was installed on our Ubuntu Server. And finally, once our installation is complete, we'll do another knife node list and now we can see that we have two registered nodes against our Chef Server. Using knife, we know that our nodes have registered successfully with Chef Server. But, let's go across to the Management console and see what the results of that registration actually are. If we head over to the Nodes tab, we can see that we now have two new registered nodes. Each of them are displaying a number of attributes like platform, fully-qualified domain name, IP address, uptime, check-in, and so on. These attributes, plus very detailed attributes, which you can access from the, well, from the Attributes tab, have all been generated by a tool called Ohai. Ohai is a component of the Chef agent and it's used to detect attributes of any server which it is installed on. It's a critical component of Chef agent because those attributes are then used by Chef admins in their everyday administrative work. For example, if you want to generate a recipe which is only applied to machines that are running a certain version of Windows, you can use the ChefDK to build up a dynamic list of servers which match a particular attribute and then assign a run list. This gives you some incredibly flexible and very, very powerful ways of managing your infrastructure. It's also possible using the ChefDK to override Ohai attributes with your own custom values or generate your own custom attributes.

Configure Clients with Chef Recipes

Now that the Chef Client has been installed successfully on both of our nodes and both nodes are registered properly with Chef Server, we're going to perform a client run. This is where each node talks back to Chef Server and assesses whether or not there are any existing or updated run lists which have been applied to it. Agent then processes those run lists and implements the recipes, which have been assigned to the node. So, we're going to launch a run and then on each of our servers, we will monitor that run and then take a look at the results of each run. The first node which we will trigger our Chef Client run on is our Ubuntu Server. As before, it uses your PowerShell to extract the fully-qualified domain name of the public IP address, which has been assigned to the Linux Server. Once we've got that put that into PuTTY and SSH into the server. The command line is very basic. It's simply Chef-Client. The client must run with elevated rights, so we specify sudo. As expected, the Chef Client picks up that the server has an empty run list. This means that there are no cookbooks or recipes which have been assigned to run against this particular server. As a result, the run makes no changes and it's over in a very short space of time. Next, we'll perform a Chef Client run on our Windows Server. Use Azure PowerShell to grab the fully- qualified domain name of the external IP and then IDP into the server. When the Chef Client was installed on the Windows Server, it was installed into C:/opscode. In order to access Chef Client functionality from the command line, we need to either go directly into that folder or, as we can see, when we take a look at the environment path, C:/opscode/chef/bin has been already added into the path list for us. The command is the same as Linux: chef-client, except that in this case, it's launching a batch file. As with our Linux Server, the run list, which has been assigned to the Windows Server, is empty. So, no cookbooks are going to run this time around. Now we verified that the Chef Client is talking back properly to Chef Server and it is looking for a run list. Now, there's not much that's very exciting about an empty run list, so let's go and put something in. So that we can see the results of all our Chef work in action, we're going to configure our Linux and Windows machines using recipes, which we created earlier on. We're going to assign each machine to a relevant role, either lab-linux or lab-windows. Remember that each of those roles has a cookbook and a recipe applied to it. We're then going to manually trigger another Chef Client run so that each node talks back to Chef Server and picks up the recipe. We're then going to see whether or not that recipe executed properly and check the results on each of our servers. Unlike the traditional security group model where we add a server to become the member of a group and then we apply attributes at the group level, when we're working with Chef roles, we add those to a run list. We could also add a recipe or a cookbook. The run list is a property of the node, so when we build up our knife command, that's what we are referencing. We use knife node to specify that we are working with a node, then run list to say that we are working with that node's run list, and then we will add it for our nominated node, - in this case, it's our Linux machine - and we specify what we want to add to the run list, which in this case, is the role lab-linux. We will then get a confirmation that this has been done and we'll run exactly the same command and just modify it slightly for our Windows machine. Again, let's jump back into the Management console to see what the impact of this has been. In the Nodes tab, select the drop-down menu underneath Actions and select Edit Run List. Here, we can see the available roles and the available recipes and which of those have been added to the node's current run list. So we can get a very easy snapshot of what run list has been applied to that particular node at that time. You can also edit the node's run list using the Management UI. Simply drag and drop roles or recipes from one side to the other. The first machine that we're going to run our recipe against is our Linux Server. So, let's switch back across to our Open SSH window. Again, we'll type in sudo chef-client, and this time, it's going to pick up the fact that there is a run list applied to it and it's going to resolve the cookbooks, which are part of that run list. In this case, lab-linux with the recipe Apache. The Chef agent is going to attempt to install the package, which we specified, but it's going to come up with an error. If we scroll up to take a look at that error, we can see that the package that we specified is not explicit enough for Chef to action it appropriately. What Chef is basically telling us is, Thank you for providing a recipe. Unfortunately, it's vague, so I'm not going to do anything. What we need to do in order to resolve this is to go back and modify our recipe. So, let's cut back across to our administrative workstation and go and open up the Apache recipe in our lab-linux cookbook. The actual package that we want to specify is Apache 2. So, we'll make the necessary modification to our recipe and save it. Now, best practice says that we should also modify the version of our cookbook because we have gone in and made a change from the version, which has been previously uploaded. It's only a very minor change, so we'll just make it 0.1.1. Save everything, go back out to the ChefDK, and use knife cookbook upload to upload our newly modified cookbook. Note that knife is reading the metadata file and the version number is the updated one. If we then do a list, we can see that the version of the cookbook, which is on the Chef Server, has been updated appropriately. Now, we can go back across to our Linux Server and we'll give that another go. So, "sudo chef-client" and this time, we receive no errors. Chef has enough information to action the changes which we specified, i.e. to install Apache 2, and it will tell us which version of that package has been installed. So that we can verify that Apache is now effectively up and operational, let's do "sudo service Apache 2 status," and we get the result that Apache 2 is running. So, it looks like our Chef recipe has been actioned properly. The ultimate test, of course, is to see whether we can hit the Apache web server from an external browser. So, use Azure PowerShell to pull out the external name of our Linux Server and put that name into a browser. And, awesome. Everything works. That means our Chef recipe functioned exactly as we wanted it to. Once we fixed it, of course. For our next trick, let's try the same process with our Windows Server. So, switch across to the RDP session, which we already have open and launch another Chef Client run using chef-client.bat. As with a Linux Server, the Windows Server will see that there is now a run list that has been applied to it. It'll find the relevant cookbooks and then it will download all of the resources it needs in order to action them. It will then attempt to perform the actual recipe, which we specified, which is to enable Telnet. Unfortunately, we don't get pretty colors to tell us that something has gone wrong, but if we look at the output of the run, we can see that there were errors and that Telnet was not installed. The exact error is that Telnet-client is unknown. This is a little strange because that is the right name for this feature. However, what we can see is that Chef attempted to use the Deployment Image Servicing and Management Tool to install Telnet. This gives us an idea to what went wrong. The feature name Telnet-client is the name that we would use if we were using PowerShell to install the feature. DISM is a different tool. It doesn't use PowerShell and its naming convention is slightly different. If we wanted to use DISM to perform the installation, then the feature name would need to be Telnet Client without the dash. So again, our recipe isn't exactly right. We have a couple of options. We can either go back into our recipe and change the name of the feature or we can make use of the fact that the Windows cookbook - and remember that the Windows cookbook is the one that we needed as a dependency of our lab-windows cookbook - and we can make use of the fact that this cookbook has multiple providers. A provider is a method that Chef can use in order to get the job done. By default, when we use the Windows cookbook to enable a feature, it uses DISM. But we can tell it to use PowerShell. So, let's cut back across to our lab-windows cookbook and take a look at the default recipe. We'll leave the feature name the same. We'll type in all true. That means that if Chef is prompted by the underlying provider, such as PowerShell, whether it's sure it wants to perform a certain action, Chef will always say, Yes. And then, we will specify the provider that we want to use. In this case, windows\_feature\_powershell. As before, we'll go into our metadata file and we'll amend the version to 0.1.1. We'll then use knife to upload the updated cookbook and we'll do a knife cookbook list in order to make sure that it has been updated properly. Now, we can cut back across to our Windows Server and we'll perform another Chef Client run. This time, we receive a PowerShell overlay, showing that there have been no errors found in the recipe and that Chef is performing the action which we asked for. Once the Chef Client run is done, we can see that the output is that the Telnet client has been installed. However, to verify it, we will check ourselves by using PowerShell and use a get-windows feature to ensure that it has actually been installed. And so, we have now successfully configured both our Linux and our Windows Servers using our Chef recipes. Now, there's nothing more we need to do and we are very nearly done, but before you pop the champagne and go dancing down the street, there's one thing we should take a look at. Think back to earlier in the course where we installed all the Chef Server components. One of the ones that we installed was Chef Reporting or Opscode-reporting. We haven't taken a look at that yet because there were no Chef Client runs to report any data to. Now that we've performed a number of runs, that information should be contained within Chef Reporting for us to take a look at. So, let's cut back to the Management console and go to the Reports tab. This time, instead of an empty dashboard, we have a run summary showing us what percentage of Chef runs worked and what percentage of them failed. If we delve down into the run history, we can get an in-depth analysis of every single run: When it occurred, which node performed it, and what the result was. Remember that we were reading the output of the Chef Client run within the console or the command window of each managed node. Instead of that, we could've just come to the reporting nodes and checked out the error log. All of the information we needed to troubleshoot our Chef Client runs is contained here. As you use Chef across more and more nodes, even though we're using knife for our everyday administrative tasks, there is still value in having the Management console available for reporting information like this. And as I promised, we don't need to actually do anything and we are properly done. Now that we've reached the end of the module, let's do a quick recap. We provision client servers by extending our Chef lab environment using Azure Resource Manager. We then connected to those servers so that we could bootstrap the Chef Client onto them. We executed a client run and took a look at the results. And then, because the results were frankly boring, we wrote a couple of recipes, cookbooks, uploaded everything into Chef, and we changed our client servers using those cookbooks. And, congratulations, you've reached the end of the course. Let' take a wander down memory lane all the way back to Module 1. We built a lab in Azure using Azure Resource Manager, infrastructure as code. Even though we had the option of going for hosted Chef or a preconfigured Chef Server in Azure, we built our Chef Server from scratch. We then installed and configured the ChefDK on our administrative workstation and used that to write some recipes and cookbooks. We provisioned some extra servers and bootstrapped the Chef Client onto them, then using our infrastructure as code, we told Chef what we wanted those servers to look like. We executed runs against them, fixed any problems, and ended up with the result that we wanted. And again, congratulations on making it through to the end of this course. It's been an absolute pleasure presenting to you on Chef and I hope we meet again in another Pluralsight course. My name's James Bannan. See you later.