Lesson 5: Calm Automation for Three-Tier Web Application with Lifecycle Management

Table of Contents

[Lesson Overview 2](#_Toc46999843)

[The LAMP Stack 4](#_Toc46999844)

[Linux 4](#_Toc46999845)

[Apache 4](#_Toc46999846)

[MySQL 5](#_Toc46999847)

[PHP 5](#_Toc46999848)

[Advanced Calm Actions 6](#_Toc46999849)

[Additional Create Stages: Pre-create and Package Install 6](#_Toc46999850)

[Additional Delete Stages: Package Uninstall and Post-Delete 6](#_Toc46999851)

[Scale-in and Scale-out 6](#_Toc46999852)

[Custom Actions for Maintenance and Operations 7](#_Toc46999853)

[Calm Orchestration and Macros 8](#_Toc46999854)

[Orchestration Dependencies across Services 8](#_Toc46999855)

[Load Balancers 8](#_Toc46999856)

[Designing the Load Balancer Tier 8](#_Toc46999857)

[Working with Multiple Application Profiles 8](#_Toc46999858)

[Reference Outline 9](#_Toc46999859)

# Lesson Overview

In this lesson, the final lesson of Course 2, we’re going to bring everything together. So far, we have focused on deepening your understanding of the various components of Calm.

We began by discussing Prism Central, and how some Prism Central capabilities tie directly into Calm. Examples of these inter-related features are:

* How Prism Central RBAC feeds into Calm roles
* How image management in Prism supports service definitions in Calm

Then we moved on to Calm itself. We explored several major Calm features and took a close look at their UI elements, as well as the ways in which you would typically interact with these features. So far, we’ve gone into detail about:

* Providers: Providers are cloud service providers, bare-metals, or existing machines that you can use to deploy, monitor, and govern your applications. Essentially, configuring a provider provides the required authorization for Calm to manage your applications using the provider’s virtualization resources.
* Projects: A project defines a set of Active Directory groups with a common set of requirements or a common structure and function, such as a team of engineers collaborating on a product. The project also specifies the roles to associate with its members, networks that they can use, infrastructure to deploy onto, and (optionally) usage limits on infrastructure resources. You can also define the environment associated with a project, in case you want to publish the applications into the Marketplace.
* Role-Based Access Control (RBAC): RBAC lets you define different roles in an organization and assign permissions accordingly. Within Calm, RBAC enables organizations to control who can perform specific actions.
* Blueprints: Blueprints are recipes for applications. These recipes encompass application architecture and Infrastructure choices, provisioning and deployment steps, application binaries, command steps, monitoring endpoints, remediation steps, licensing and monetization, and policies. Every time a Blueprint is executed it results in an application deployment, these workloads can be managed from the Applications menu.
* The Marketplace: The Marketplace is a common platform for both the publisher and the consumer, and provides you with the ability to provision an application instantly. It provides a set of pre-seeded application Blueprints that are available for you to use.
* Application Profiles: Application Profiles expose simple choices to your end users. These choice are often about where an application should run (AHV or AWS). They can also be used for ‘t-shirt’ sizing (small or large), or a combination of location and sizing (small AHV or large AHV or small AWS).
* Services: Services are logical entities exposed by an IP that span all application profiles and are managed by Calm. End users and services communicate with each other over a network using their exposed IPs and ports.
* Substrates: Substrates are combinations of the underlying cloud and the VM instance. When you select the desired cloud in the Calm UI, all the fields required to create a VM instance on that particular cloud are displayed. The combination of these fields is a substrate.
* Service Actions: Services Actions are a set of operations that you can run on your application. For example, when launching a blueprint, the ‘Create’ action is run. If your application is not needed for a period of time, you could then run the ‘Stop’ action to gracefully stop your application.
* Variables: The properties such as IP addresses, DNS names, and instance IDs that are associated with the services provisioned in blueprints are called variables. They can be static, provided at run time, or generated during blueprint or action runs.
* Macros: Macros enable you to access the value of variables and properties set on entities, and help you make generic scripts and create reusable workflows.

Each of these features represent essential elements of Calm’s development workflow and overall application lifecycle.

And in this lesson, we are going to bring all of this knowledge together. We are going to use Calm’s various capabilities to create a web server application and understand — from a real-world development perspective — what Calm is really capable of.

To achieve this, we will be discussing:

* The LAMP stack, which is the foundation on which we will be building our application
* Calm Actions, in a little more detail than we have in previous lessons
* Scale-in and scale-out web services
* Calm Orchestration and Macros, as well as orchestration dependencies across services
* Load balancers and how to design the load balancer tier
* Working with multiple application profiles

Let’s begin.

# The LAMP Stack

LAMP is an acronym which expands to Linux, Apache, MySQL, and PHP. It is the most common example of a web service stack, and is popularly used to build dynamic website and web applications.

Over time, LAMP has become a term used to refer to a generic software model, and the model itself has been adapted to include different components. Two common examples are:

* A variant in which Linux is replaced by Windows, abbreviated as WAMP
* A variant in which Apache is replaced by Internet Information Services (or IIS), amusingly abbreviated as WIMP

The original LAMP stack continues to remain popular, however, largely for its flexibility and because it is capable of hosting a variety of web frameworks. For example, if you are familiar with WordPress, Joomla, or Drupal, you may be interested to know that they run on the LAMP stack.

Owing to both its flexibility and popularity, you are likely to encounter the LAMP stack when working with web applications — which is why we are going to build our three-tier Calm web app on the LAMP stack as well.

But before we go into more detail, let’s talk about the LAMP stack itself.

## Linux

Linux belongs to the family of Unix-like operating systems. It was written by Linus Torvalds and has the features that are typical of a modern Unix OS, including multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management, and multistack networking including IPv4 and IPv6.

To use Linux you need to download a distribution, which is a complete Linux system including the kernel and applications. Multiple distributions are available for download and, as you may remember, one that we discussed in an earlier lesson is CentOS.

In the LAMP stack, Linux is the foundation. Everything else runs on top of this layer.

## Apache

The second layer, the web server, is typically the Apache HTTP Server but can also be IIS or Nginx. Apache is used simply because it is a mature, feature-rich product and is arguably the most popular web server on the Internet.

Apache HTTP Server (commonly shortened to just ‘Apache’) is a free, open-source, cross-platform web server that played a pivotal role in the initial growth of the World Wide Web. It overtook NCSA HTTPd, has remained popular since 1996, and became the first web server to host more than one hundred million websites.

Apache’s features include Secure Sockets Layer (SSL) and Transport Layer Security (TSL) support, proxying, large file support, custom log files, and so on. Many features are also implemented as complied modules, which extend the core functionality of the web server. These include authentication, authorization, support for server-side programming languages such as Perl and Python, and so on.

## MySQL

The third layer is the database layer and has traditional been filled by MySQL. Popular alternatives include MariaDB and PostgreSQL, as well as NoSQL databases such as MongoDB.

MySQL is a free, open source relational database management system (RDBMS). As with any relational database, it organizes data into one or more data tables, in which the data types are related to each other. These relations help structure data. And from the perspective of the LMAP stack, MySQL stores data that, when queried via scripts, can be used to construct a website.

## PHP

The fourth and final layer of the LAMP stack is the application programming language. While this is commonly PHP, the role can be filled by other languages such as Perl and Python.

PHP was created in 1994 by Rasmus Lerdorf, who wrote a number of common gateway interface programs in C. He extended these programs to work with web forms and communicate with databases, and called this implementation Personal Home Page/Forms Interpreter — which is abbreviated as PHP/FI.

PHP is a general purpose scripting language that is especially useful for web development, and is also used as a general purpose programming language. PHP code is interpreted by a web server via a PHP processor module, which generates the resulting web page.

# Advanced Calm Actions

As we’ve discussed in previous lessons, an action is a set of operations that you can run on an application or a service. There are two major types of actions: Profile Actions and Service Actions.

Profile Actions are run on applications and are of two types: system-defined profile actions, and custom profile actions. System-defined profile actions are automatically created by Calm in every blueprint and underlying application. These actions are typically Create, Start, Stop, Delete, and so on. Custom profile actions are created by the person developing the blueprint, and are typically Upgrade, Scale In, Scale Out, and so on.

Service Actions are run on individual services and typically span application profiles. So, for example, consider that you have two profiles set up in a blueprint, AWS and AHV. If you create a service action for the AHV profile, it will be available in the AWS profile as well. As with profile actions, there are two types of service actions: system-defined and custom.

For this lesson specifically, we need to take a closer look at how Calm actions can handle more complex PaaS and SaaS needs. This is typically achieved through:

* Additional Create stages
* Additional Delete stages
* Scale-in and Scale-out
* Custom actions for ongoing maintenance and operations

## Additional Create Stages: Pre-create and Package Install

Pre-create actions are run before the substrate is created. A common use case for this is to make an API call into an IP Address Management (IPAM) system to get an IP for a to-be-created VM.

Package Install actions, on the other hand, are operations which are run during the Create profile action. In other words, they are operations that run when a user first launches a blueprint. Package Install is unique to each application profile, which means your tasks or the task contents can vary depending upon the underlying cloud or the app’s size.

## Additional Delete Stages: Package Uninstall and Post-Delete

Package Install actions are operations which are run during the Delete profile action. That is, they are operations that run when a user finally deletes the entire application. Like Package Install, Package Uninstall is also unique to each application profile, so tasks or task contents can vary depending upon the underlying cloud or app size. An example of a package uninstall task might be to cancel or release a license entitlement when the software is removed from the service or to orchestrate dependent changes on other services required for the uninstall process.

Post-delete actions are run after the substrate is deleted. For example: when a VM is deleted, the post-delete action would trigger removing the VM server property held by external systems. Typical use cases would involve deprovisioning the IP, DNS, and/or MAC addresses of VMs with various different inventory, monitoring, or change management database (CMDB) systems. For example: when a VM is deleted, the post-delete action would trigger removing the VM server property from any of the following:

* + Retire the IP address from the IPAM system, such as Infoblox, BlueCat, Redhat Satellite, etc.
  + Delete all associated DNS records from the DNS provider, such as Bind, PowerDNS, AWS Route53, etc.
  + Remove VM from asset inventory in ServiceNow.

In other words, post-delete action and package uninstall tasks help clean up and remove everything necessary outside of the application services when an application is deleted.

## Scale-in and Scale-out

Although we’ve discussed these topics briefly in a previous lesson, let’s take a moment to recap them both here.

Scale-in functionality enables you to decrease the number of replicas of a service deployment. The number of instances to be removed from a service for each scale-in action is defined in the blueprint. This is typically done when configuring the task in the profile action. The scale count number must be less than or equal to the minimum number of replicas defined for the service. The VM that is created last is deleted first.

Scale-out functionality, as you may have guessed already, is the opposite of scale-in — enables you to increase the number of replicas of a service deployment. The number of instances to be added to a service for each scale-out action is defined in the blueprint, when configuring the task in the profile action. The scale count number must be less than or equal to the maximum number of replicas defined for the service.

## Custom Actions for Maintenance and Operations

As you’ve learned from the Create action, complex orchestration across the entire application services can be configured during deployment of the application and infrastructure. We’ve also covered the scaling actions for changing a running application’s replica services. How could one perform any other change to a running application?

Custom actions can be created in Calm blueprints by the developer role for post-deployment or “day two” operations, to allow any simple or complex operation to be orchestrated during the running state of the application. Aside from the existing start/stop/restart and delete actions, custom actions allow life cycle operations and updates after the birth, but before the death or termination of the application workload. Here are some typical maintenance examples:

* add/update/reset a user credential to a service, e.g.: add a new database user
* add/update/reinstall a new version of a package, e.g.: a web application update
* add/update a configuration and reload a service, e.g.: a web server change
* backup/restore a VM disk or service, e.g.: a database backup
* update an operating system, e.g.: install security updates
* any combination of the above

Because these custom actions can be delegated to any user or group in the project with Operator role or higher, these auditable actions can be made self-service for end users, freeing valuable time to IT and Operations staff. Even better, periodic scheduling systems or monitoring systems can detect problems can trigger these custom actions for automated, continuous operations via Calm APIs to allow “lights-out” remediation and maintenance of applications around the clock.

Reference: An intriguing example using a Prism Central advanced feature to trigger Calm custom actions: [Private Cloud X-Play + Calm](https://www.youtube.com/watch?v=M5tTI25NcXo) YouTube video of 9:46.

# Calm Orchestration and Macros

By default, tasks are executed sequentially on each service for each action in a blueprint and each service will operate in parallel, fully independent of each other. However, this is rarely useful for multiple services: they often work together and require operational coordination. Orchestration is how Calm drives the order of operations across all services in the blueprint.

Calm automatically determines orchestration dependencies between services when a task in an action uses a macro value from another service in the blueprint. An orchestration dependency is represented by an orange arc pointing from the dependent task to the other service task which would populate the macro value. For example, we saw that the load balancer service configuration depended on knowing each web server IP address using the web service address macro.

If needed, you can also manually create or delete a dependency arc between any two tasks of any service in a blueprint for very specific orchestration requirements, which is an advanced topic outside of our lesson.

Calm will validate all dependency arcs when a blueprint is saved and alert if there are any cyclical or ambiguous errors. This is why you may not see dependencies during the creation or editing of a multi-VM blueprint until saving the blueprint.

# Orchestration Dependencies across Services

If you do not use macros in your blueprint for automated orchestration as covered in the previous section, a simpler method exists to allow logical orchestration across multiple services during all of the standard actions (create, start, stop, restart, delete). Simple use the white arc to connect one service to another, starting from the dependent and ending on the source of the dependency.

When you save the blueprint, you will see orchestration dependency arcs created for each of the actions, even if each action has no tasks. Furthermore, you will notice that the orchestration is reversed in create versus delete and start versus stop actions.

* References: <https://portal.nutanix.com/page/documents/details?targetId=Nutanix-Calm-Admin-Operations-Guide-v3_0_0:nuc-components-dependencies-overview-c.html>
* <https://portal.nutanix.com/page/documents/details?targetId=Nutanix-Calm-Admin-Operations-Guide-v3_0_0:nuc-setting-up-service-dependencies-t.html#ntask_xhm_nxk_b1b>

# Load Balancers

Load Balancers are a critical architectural function in modern infrastructure and application design, they serve as a front-end for a service and distribute “load” across service resources. In our lesson, we’ll use a load balancer to split web traffic across the network to a web service array. Often, you will hear the terms request, load, traffic, and transaction used interchangeably, they have very close meanings that may be used separately in each service’s terminology.

For our simple needs, our load balancer will use the simplest distribution mechanism called “round robin.” A web request from a web client is directed to the load balancer, which picks one web server from the array for the first web request and delivers the web request to it. The web server answers the request and returns the answer through the load balancer back to the web client requestor. For the next web request to the load balancer, it will pick the next web server available in the array. This repeats for each subsequent web request across the entire web server population and will cycle around to the beginning of the web server array.

The load balancer allows flexibility for web operations:

* Performance: distributes web requests, allowing us to scale in and scale out the web tier as transactions grow more complex or resource hungry.
* High uptime: allows removal of an individual web server for maintenance and return once complete while the remaining web servers handle traffic.

Contrast the above benefits to previous strategies to scale up the resources of a single web server to handle more traffic, but this has definite physical limits to scale combined with increasing expense for more CPU and memory and incurs downtime for maintenance.

# Designing the Load Balancer Tier

We’ll use a very popular, open source load balancer called HAProxy, which is included in many Linux distributions, including our choice of CentOS. We’ll install HAProxy and configure it to accept web connections on TCP port 80. Furthermore, we’ll configure HAProxy to distribute requests to the web server array specified using Calm macros for the web service array.

Reference: <https://en.wikipedia.org/wiki/HAProxy>

# Working with Multiple Application Profiles

MARK: Let us move this to Course 3 unless we feel Course 2 lesson 5 is too light. ☺ I’ll make this adjustment later this week.

<Again, I’m having trouble understanding the context of this section, so I’m not sure what to include here from a concept point of view.>

Application Profiles expose simple choices to your end users.  Often these choices are on where an application should run (AHV or AWS), but they can also be used for “t-shirt” sizing (Small or Large), or a combination of the two (Small AHV or Large AHV or Small AWS).  As an IT operator or developer, you should have a good grasp on the underlying differences of these choices, while abstracting that complexity from your end users.

An application profile provides different combinations of the service, package, and VM while configuring a blueprint. If Showback feature is enabled, the application profile also displays service cost of the resources used for an application. You must select an application profile while you launch a blueprint. The application profile also enables you to use the same set of services and packages on the different platforms.

* <https://portal.nutanix.com/page/documents/details?targetId=Nutanix-Calm-Admin-Operations-Guide-v3_0_0:nuc-adding-configure-application-profile-t.html#ntask_hjs_xxk_b1b>

# Reference Outline

<https://github.com/mlavi/calmbootcamp/blob/master/udacity/hce/2/outline.md>