Basic Automation Scripting

7.3.1

Introduction to Basic Automation Scripting

Powerful automation tools like Ansible, Puppet, and Chef bring ease of use, predictability, discipline, and the ability to work at scale to DevOps work. But that does not mean that you cannot do some automation with more basic tools like Bash and Python. Automation tooling partly works by wrapping shell functionality, operating system utilities, API functions and other control plane elements for simplicity, uniformity, feature enrichment, and compatibility in DevOps scenarios. But tools still do not solve every problem of deployment and configuration.

That is why every automation tool has one or more functions that execute basic commands and scripts on targets and return results. For example, in Ansible, these functions include `command`, `shell`, and `raw`.

Sometimes it can be faster and simpler to use shell commands or scripts. Often, this is because many tool implementations begin by translating automation originally written in Bash, Python, or other languages, and you want to transfer that functionality quickly and accurately into the tool before porting and refactoring.

In summary, it is rare to look deep down into tool-maintained infra-as-code repos without finding some scripting. So having these skills is important!

7.3.2

Basic Tools for Automation Scripting

Shells are ubiquitous, so shell scripting is historically the bedrock of automation.

Bash

In Linux (and other operating systems) the shell interoperates with interactive I/O, the file system, and interprocess communication. This provides ways to issue commands, provides input for processing and piping outputs to chains of powerful utilities.

The Bourne Again Shell (BASH) is the default on most Linux distributions. Because of its ubiquity, the terms "Bash" and "shell" are generally used interchangeably.

Using commands in a Bash script is much the same as using them directly from the command line. Very basic script development can simply be a matter of copying command-line expressions into a file after testing the CLI commands to see if they work.

By contrast, Python or another high-level, sophisticated language for simple procedural automation is usually more challenging, and may not be worthwhile for simple projects.

Programming languages beyond Bash

Sophisticated languages improve on Bash when complexity and scale requirements increase. They are particularly useful when building and operating virtualized infrastructure in cloud environments, using SDKs like the AWS SDK for Python or the AWS SDK for javascript in Node.js. While Bash can be used to script access to the AWS CLI, you can use the built-in features and libraries of more sophisticated languages to parse complex returned datasets (such as JSON), manage many parallel operations, process errors, handle asynchronous responses to commands, and more.

To develop and execute scripts in your desired language, you may need to install and configure that language on your development system and on any remote target machines. Accessing system-level utility functions may require invoking libraries (such as the `os` library in Python), then wrapping what are Bash CLI commands in additional syntax for execution. You also need to handle return codes, timeouts, and other conditions in your preferred language environment.

7.3.3

Procedural Automation

Using Bash, Python, or other conventional languages for automation usually means writing an imperative procedure. An imperative procedure is an ordered sequence of commands aimed at achieving a goal. The sequence may include flow-control, conditions, functional structure, classes, and more.

Such procedural automation can be very powerful. But it stays simple only if you are knowledgeable about how system utilities, CLIs/SDKs, and other interfaces work. You must also know about target system state.

Developing a procedure

As you know, if you make a little script to install and configure a piece of software on a remote target system, it may run okay the first time. Run it a second time however; and your simple script might make a mess. It might throw an error and stop when it finds the application already installed, or worse, ignore such an error, and go on to make redundant changes in config files.

To make this script safer, easier to use, more flexible, and reusable, you need to make it smarter and more elaborate. For example, you could enhance it to:

- Determine if it is running in a Debian or a CentOS environment, and use the correct package manager (`apt` or `yum`) and syntax.
- Determine if your target app is already installed in an appropriate version, and only try installing it if it is not present, stopping otherwise and making no further changes.
- Determine if it has made a copy of each config file before changing it, and use stream editors (`awk`, `sed`, etc.) to make precise changes, rather than carelessly appending text to config files, and hoping the applications that consume these files will not break.

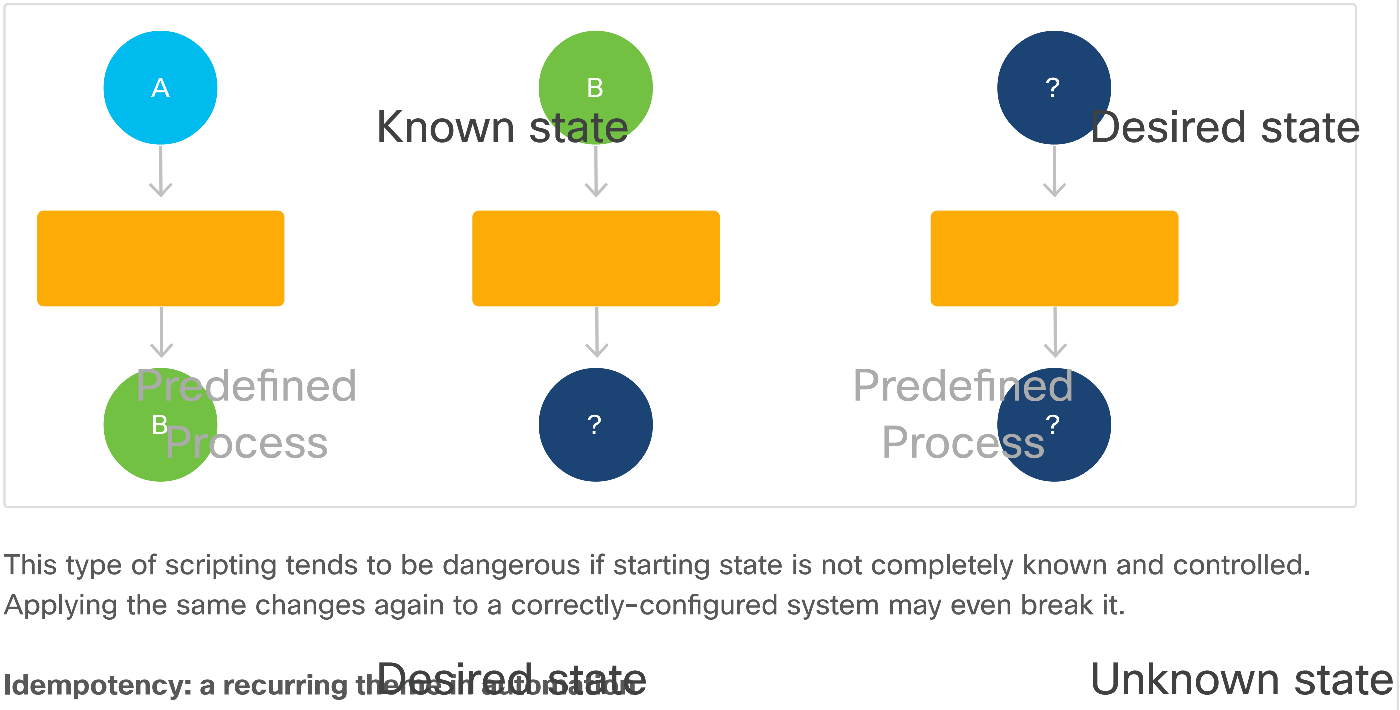
As you develop and refine the scripts further, you will want them to accomplish some of the following tasks:

- Discover, inventory, and compile information about target systems, and ensure the scripts do this by default.
- Encapsulate the complexity of safely installing applications. Make config file backups and changes, and restart services into reusable forms, such as subsidiary scripts containing parameters, function libraries, and other information.

To ensure the scripts are efficient and reusable, you will:

- Standardize the ordering and presentation of parameters, flags, and errors.
- Create a code hierarchy that divides tasks logically and efficiently.
- Create high-level scripts for entire deployments and lower-level scripts for deployment phases.
- Separate deployment-specific data from the code, making the code as generic and reusable as possible.

Simple Procedural Scripting Example



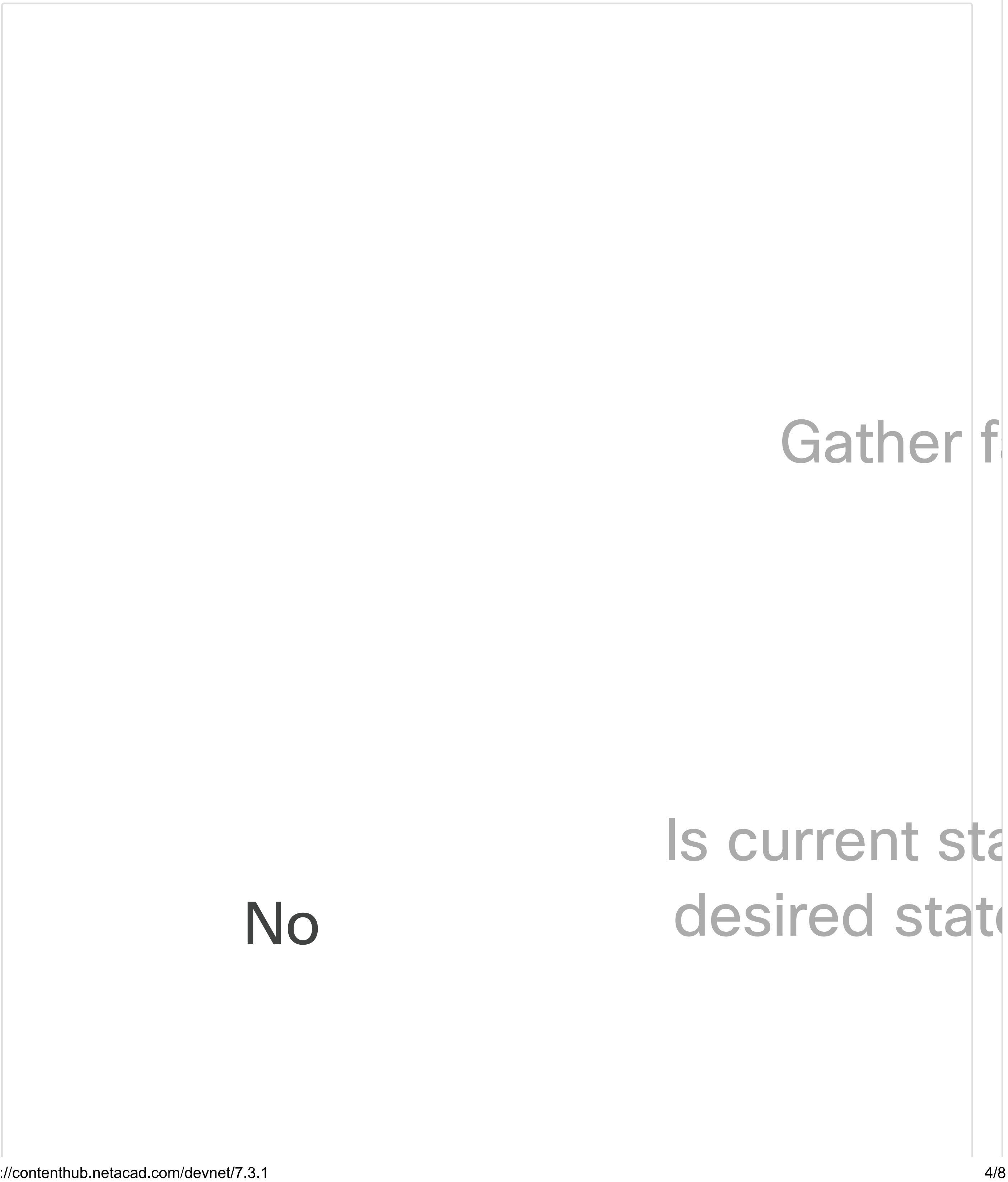
This type of scripting tends to be dangerous if starting state is not completely known and controlled. Applying the same changes again to a correctly-configured system may even break it.

Idempotency: a recurring theme in automation

Ultimately, the goal of almost any script is to achieve a desired state in a system, regardless of starting conditions. Carefully-written procedural scripts and declarative configuration tools examine targets before performing tasks on them, only performing those tasks needed to achieve the desired state.

This quality of software is called idempotency. There are a few basic principles of idempotency to follow:

- **Ensure the change you want to make has not already been made -** Also known as "First, do no harm". Doing nothing is almost always a better choice than doing something wrong and possibly unrecoverable.
- **Get to a known-good state, if possible, before making changes -** For example, you may need to remove and purge earlier versions of applications before installing later versions. In production infra-as-code environments, this principle becomes the basis for immutability. Immutability is the idea that changes are never made on live systems. Instead, change automation and use it to build brand-new, known-good components from scratch.
- **Test for idempotency**: Be scrupulous about building automation free from side effects.
- **All components of a procedure must be idempotent -** Only if all components of a procedure are known to be idempotent can that procedure as a whole be idempotent.

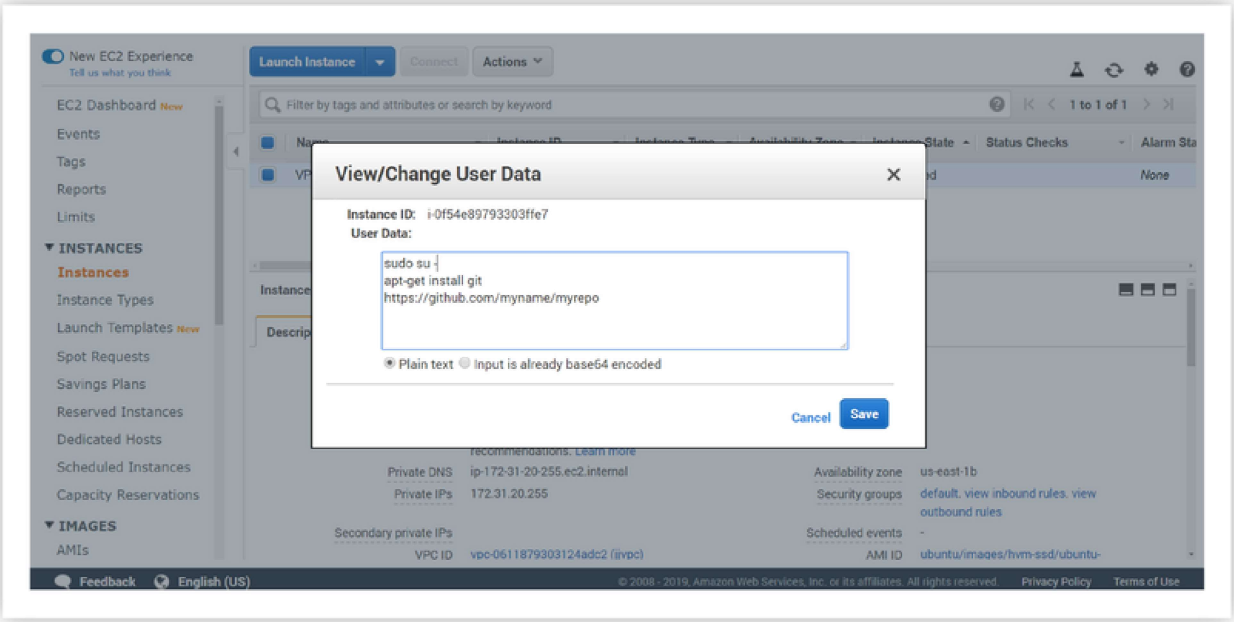


7.3.4

Executing Scripts Locall

To configure remote systems, you need to access and execute scripts on them. There are many ways to do this:

- You can store scripts locally, transmit them to t with a shell utility like `scp`, then log into the remote machine using `ssh` and ex
- You can pipe scripts to a remote machin cute them in sequence with other commands, capturing and returnin l in one command.
- You can install a general-purpose secure fil SFTP, then use that utility to connect to the remote machine, transfer, set appropriat then execute your script file.
- You can store scripts on a webserver, log into the r machine and retrieve them with `wget`, `curl`, or other utilities, or store the scripts in a Git repository. Installing git on the remote machine, clone the repo to it, check out a branch, and execute the scripts found there.
- ations solution like VNC or NoMachine locally, install its server on so installing a graphical desktop environment), transmit/copy and
- ned on a cloud framework, there is usually a way to inject a configuration script via the same CLI command or WebUI action that manifests the platform.



View/Change User Data

Instance ID: i-0f54e89793303ffe7

User Data:

```
sudo su -
apt-get install git
https://github.com/myname/myrepo
```

● Plain text ○ Input is already base64 encoded

Cancel Save

Most public cloud services let you inject configuration scripts directly into VM instances for execution at boot time

Almost every developer will end up using these and other methods at one point or another, depending on the task(s), environmental limitations, access to internet and other security restrictions, and institutional policy.

Understanding these methods and practicing them is important, because procedurally automating certain manual processes can still be useful, even when using advanced deployment tools for the majority of a DevOps task. To be clear, this is not good practice, but the state of the art in tooling is not yet perfect or comprehensive enough to solve every problem you may encounter.

7.3.5

Cloud Automation

Infrastructure-as-a-Service (IaaS) cloud computing frameworks are a typical target for automation. Cloud automation enables you to provision virtualized hosts, configure virtual networks and other connectivity, requisition services, and then deploy applications on this infrastructure.

Cloud providers and open source communities often provide specialized subsystems for popular deployment tools. These subsystems extract a complete inventory of resources from a cloud framework and keep it updated in real time while automation makes changes, which enables you to more easily write automation to manage these resources.

You can also manage cloud resources using scripts written in Bash, Python, or other languages. Such scripts are helped along by many tools that simplify access to automation targets. These include:

- CLIs and SDKs that wrap the REST and other interfaces of hardware, virtual infrastructure entities, higher-order control planes, and cloud APIs. This makes their features accessible from shells (and via Bash scripts) and within Python programs.
- Command-line tools and Python's built-in parsers can parse JSON and YAML output returned by CLIs and SDKs into pretty, easy-to-read formats and into native Python data structures for easy manipulation.

7.3.6

Cloud CLIs and SDKs

IaaS and other types of infrastructure cloud also provide CLIs and SDKs that enable easy connection to their underlying interfaces, which are usually REST-based.

Cisco UCS - a bare metal cloud

If you are familiar with Cisco Compute products, including Unified Computing System (UCS), Hyperflex, UCS Manager, and the Intersight infrastructure management system, you know these are effectively a gateway to global SaaS management of an organization's UCS/Hyperflex infrastructure.

Cisco's main API for this infrastructure is the Cisco Intersight RESTful API. This is an OpenAPI-compatible API that can be interrogated with Swagger and other open source OpenAPI tools. These enable you to generate specialized SDKs for arbitrary languages and environments, and simplify the task of documenting the API (and maintaining SDKs).

Cisco provides and maintains a range of SDKs for the Intersight RESTful API, including ones for Python and Microsoft PowerShell. They also provide a range of Ansible modules.

VMware

VMware's main CLI is now Datacenter CLI, which enables command-line operation of vCenter Server API and VMware Cloud on AWS. It is written in Python and runs on Linux, Mac, and Windows.

VMware also provides vSphere CLI for Linux and Windows, which lets you manage ESXi virtualization hosts, vCenter servers, and offers a subset of DCLI commands. It also offers PowerCLI for Windows PowerShell, which provides cmdlets for vSphere, vCloud, vRealize Operations Manager, vSAN, NSX-T, VMware Cloud on AWS, VMware HCX, VMware Site Recovery Manager, and VMware Horizon environments.

OpenStack

The OpenStack project provides the OpenStack Client (OSC), which is written in Python. The OSC lets you access OpenStack Compute, Identity, Image, Object Storage, and Block Storage APIs.

Installing the command-line clients also installs the bundled OpenStack Python SDK, enabling a host of OpenStack commands in Python.

OpenStack Toolkits are also available for many other popular languages.

7.3.7

Summary of Basic Automation Scripting

Summary

Basic automation scripting techniques are great to have in your toolbox, and understanding them will improve your facility as an operator and user of mature automation platforms.

7.2
DevOps and SRE

7.4
Automation Tools