Nodes, Modu
Collection
Tasks, Play
Configurat
Automation
Commands, C
Console, YA
Modulariza

Ansible

The Practical Guide for Administrators and DevOps Teams

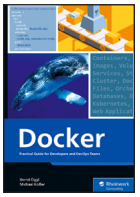
Axel Miesen

Rheinwerk Computing

Rheinwerk Computing

The Rheinwerk Computing series offers new and established professionals comprehensive guidance to enrich their skillsets and enhance their career prospects. Our publications are written by the leading experts in their fields. Each book is detailed and hands-on to help readers develop essential, practical skills that they can apply to their daily work.

Explore more of the Rheinwerk Computing library!



Bernd Öggl, Michael Kofler

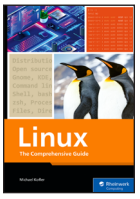
Docker: Practical Guide for Developers and DevOps Teams

2023, 491 pages, paperback and e-book
www.rheinwerk-computing.com/5650

Kevin Welter

Kubernetes: Practical Guide for Developers and DevOps Teams

2024, 401 pages, paperback and e-book
www.rheinwerk-computing.com/5964



Michael Kofler

Linux: The Comprehensive Guide

2024, 1178 pages, paperback and e-book
www.rheinwerk-computing.com/5566

Sujeevan Vijayakumaran

DevOps: Frameworks, Techniques, and Tools

2025, 418 pages, paperback and e-book
www.rheinwerk-computing.com/6030

Johannes Ernesti, Peter Kaiser

Python 3: The Comprehensive Guide

2022, 1036 pages, paperback and e-book
www.rheinwerk-computing.com/5566

www.rheinwerk-computing.com

Axel Miesen

Ansible

The Practical Guide for Administrators and DevOps Teams

Rheinwerk Computing

Imprint

This e-book is a publication many contributed to, specifically:

Editor Rachel Gibson
Acquisitions Editor Hareem Shafi
German Edition Editor Christoph Meister
Translation Torsten T. Will
Copyeditor Doug McNair
Cover Design Graham Geary
Photo Credit iStockphoto: 1190896641/© Stefan Tomic; 1534842661/© JuSun
Layout Design Vera Brauner
Production E-Book Kelly O'Callaghan
Typesetting E-Book III-satz, Germany

We hope that you liked this e-book. Please share your feedback with us and read the Service Pages to find out how to contact us.

Library of Congress Cataloging-in-Publication Control Number: 2025009054

© 2025 by:
Rheinwerk Publishing, Inc.
2 Heritage Drive, Suite 305
Quincy, MA 02171
USA
info@rheinwerk-publishing.com
+1.781.228.5070

Represented in the E.U. by:
Rheinwerk Verlag GmbH
Rheinwerkallee 4
53227 Bonn
Germany
service@rheinwerk-verlag.de
+49 (0) 228 42150-0

ISBN 978-1-4932-2702-0 (print)
ISBN 978-1-4932-2703-7 (e-book)
ISBN 978-1-4932-2704-4 (print and e-book)

1st edition 2025
1st German edition published 2025 by Rheinwerk Verlag

Contents

Preface 21
About This Book 23

1 Introduction and Installation 27

1.1 What Is Ansible? 27
1.1.1 Basic Features 27
1.1.2 Core Components 28
1.1.3 Declarative or Imperative? 28
1.2 What Is Ansible Not? 29
1.3 History and Versions 30
1.3.1 Versions 30
1.3.2 Supported Python Versions 31
1.4 Setup/Lab Environment 31
1.5 Ansible Installation on the Control Host 36
1.6 Installation via PIP (plus virtualenv) 38
1.7 Authentication and Authorization on Target Hosts 39
1.8 Setting Up SSH Public Key Authentication 41
1.9 An Ad Hoc Test without Any Configuration 42
1.10 Another Note on Migrating from Older Versions 43

2 Basic Setup and Initial Inventory Management 45

2.1 Setting Up the Directory Structure 45
2.2 Basic Configuration (ansible.cfg) 46
2.3 Creating and Managing a Static Inventory 48
2.4 Configuration Settings versus Parameters versus ...? 50
2.5 Inventory Aliases and Namespaces 51
2.6 Beyond Ping 53

2.7 A Somewhat More Complex Example ... 55
2.8 Alternative or Multiple Inventories ... 56

3 Ad Hoc Commands and Patterns 59

3.1 Ad Hoc Commands ... 59
3.1.1 The “command” Module ... 59
3.1.2 The “shell” Module ... 60
3.2 Use Cases Beyond “command” and “shell” ... 61
3.3 Idempotency ... 61
3.4 Internal Workings ... 63
3.4.1 Parallel Execution ... 63
3.4.2 Persistent Connections ... 63
3.4.3 What Happens When a Module Is Called? ... 65
3.5 The Ansible Console ... 66
3.6 Patterns for Addressing Hosts ... 67

4 YAML: The Configuration and Serialization Language 69

4.1 Syntax and Structure ... 69
4.2 Editing YAML Files ... 70
4.2.1 vim ... 70
4.2.2 nano ... 71
4.2.3 Emacs ... 72
4.2.4 Notepad++ ... 72
4.3 Syntactic Checks ... 72
4.4 Lists and Maps ... 73
4.5 Nested Structures ... 73
4.6 Text Passages and Block Expressions ... 75
4.6.1 Variations ... 75
4.6.2 Long Texts without Line Breaks ... 76
4.7 Nothingness in YAML ... 76
4.8 Anchors and References ... 77

5 Playbooks and Tasks: The Basics 79

5.1 Hello Ansible: The Very First Playbook 79
5.1.1 Relative Paths in Playbook Calls 81
5.1.2 Addressing the Actual Target Hosts 82
5.2 Formulation of Tasks 83
5.3 Ending Plays 85
5.4 The Problematic Colon 85
5.5 Error Handling and Retrying Files 86
5.6 Tags 88
5.7 The "ansible-playbook" Command 90
5.8 An Example of an Apache Installation 91
5.8.1 Step by Step 91
5.8.2 The Complete Playbook 93
5.8.3 "--start-at-task, "--check," and "--diff" 94
5.9 Handler: Executing Tasks Only on Changes 95
5.9.1 Step by Step Execution 95
5.9.2 Handlers 97
5.9.3 The Complete Playbook up to This Point 100

6 Playbooks and Tasks: Advanced Methods 103

6.1 Variables 103
6.1.1 Play Vars 103
6.1.2 Extra Vars 104
6.1.3 Precedence 104
6.1.4 "set_fact" 105
6.1.5 "group_vars" 106
6.1.6 "host_vars" 108
6.1.7 "vars_files": Variables in Any External Files 109
6.1.8 Prompting 109
6.1.9 Accessing Complex Structures 110
6.1.10 "assert" Tests 111
6.2 Registered Variables 112
6.3 Facts and Implicit Variables 115
6.3.1 Facts 115
6.3.2 Caching of Facts 117

6.3.3 Implicit Variables ... 118
6.3.4 Example ... 119
6.3.5 External Information Gatherers: “facter” and “ohai” ... 120
6.3.6 Not Enough Facts Yet? Try “/etc/ansible/facts.d”! ... 120

6.4 Conditional Execution with “when” ... 121

6.5 Handling System Differences: What’s the Deal Now? ... 122
6.5.1 The Clumsy Method ... 123
6.5.2 The Solid Method ... 124
6.5.3 The Tricky Method ... 126
6.5.4 The Modular Method ... 127
6.5.5 The Complete Playbook up to This Point ... 129

6.6 Jinja and Templates ... 131
6.6.1 Definition: Templates and Template Engines ... 131
6.6.2 An Individual Homepage for Our Apache Servers ... 132
6.6.3 Quick Testing of Jinja Templates ... 134
6.6.4 Jinja Syntax: Output Expressions, Statements, and Comments ... 136
6.6.5 Filter ... 137
6.6.6 Whitespace Control ... 138
6.6.7 Macros ... 140

6.7 Variable Tests ... 141

6.8 Lookup Plug-ins ... 143

6.9 Loops ... 145
6.9.1 “with_*” Loops Are Lookups ... 145
6.9.2 Iteration Over a List with “with_items” or “with_list” ... 146
6.9.3 Iteration Over a Map with “with_dict” ... 149
6.9.4 Iteration Over a Generated Sequence with “with_sequence” ... 149
6.9.5 Looping Over the Combination of Two Lists with “with_nested” ... 150
6.9.6 Looping Over Two Parallel Lists with “with_together” ... 151
6.9.7 Nested Loops with “with_subelements” ... 151
6.9.8 Repeating Tasks with “until” ... 152
6.9.9 More Control with “loop_control” ... 154
6.9.10 “register” plus Loop ... 157

6.10 Error Handling with “failed_when” and “ignore_errors” ... 158

6.11 Blocks ... 159

6.12 Timeouts and Asynchronous Execution ... 161

6.13 Local Tasks ... 163
6.13.1 “local_action” ... 165
6.13.2 Using Other Hosts ... 165

6.14 Environment Variables ... 166
6.14.1 Setting Environment Variables with the “environment” Parameter ... 166
6.14.2 Reading Environment Variables ... 167
6.14.3 Environment Variables in “path” or “dest” Attributes ... 168

7 Using Modules and Collections 169

7.1 Collections ... 169
7.1.1 A Minimal Environment with “ansible-core” ... 169
7.1.2 Managing Collections ... 170
7.1.3 The Fully Qualified Collection Name ... 172
7.1.4 Interim Conclusion ... 173

7.2 Module ... 173

7.3 Modules for Command Execution ... 174
7.3.1 “command”: Executing Commands ... 174
7.3.2 “shell”: Executing Commands via a Shell ... 174
7.3.3 “raw”: Executing Commands without the Python Subsystem ... 174
7.3.4 “script”: Transferring and Executing Local Scripts ... 175

7.4 Modules for Package Management ... 175
7.4.1 “apt”: Package Management on Debian/Ubuntu Systems ... 175
7.4.2 “dnf” and “yum”: Package Management on Rocky, CentOS, Red Hat, and Fedora Systems ... 176
7.4.3 “community.general.zypper”: Package Management on SUSE Systems ... 176
7.4.4 “package”: Generic Package Management ... 176
7.4.5 “package_facts”: Representing Package Information as Facts ... 176
7.4.6 Some Other (Not Further Described) Modules ... 177

7.5 Modules for Managing Files and File Contents ... 177
7.5.1 “copy”: Copying and Uploading Files ... 177
7.5.2 “template”: Processing Files with Jinja and Uploading ... 178
7.5.3 “file”: Managing Files and File Attributes ... 178
7.5.4 “stat”: Obtaining Information about Files ... 179
7.5.5 “lineinfile”: Managing Lines in Text Files ... 180
7.5.6 “blockinfile”: Managing Text Passages in Files ... 180
7.5.7 “replace”: Searching and Replacing in Text Files ... 181
7.5.8 “unarchive”: Uploading and Extracting Archives ... 181
7.5.9 Some Other (Not Further Described) Modules ... 181

7.6 Modules for Other Typical Administrative Tasks ... 182
7.6.1 “service”: Starting, Stopping, and Restarting Services, Etc. ... 182

7.6.2 “service_facts”: Presenting Service Information as Facts 182
7.6.3 “cron”: Managing Cron Jobs 183
7.6.4 “hostname”: Changing the Hostname 183
7.6.5 “user” and “group”: User and Group Management 183
7.6.6 “ansible.posix.firewalld”: Managing firewalld-Based Firewalls 184
7.6.7 “reboot”: Rebooting Machines 184
7.6.8 Some Other (Not Further Described) Modules 184

7.7 Modules for Interacting with Network Services 185
7.7.1 “get_url”: Downloading Resources 185
7.7.2 “uri”: Communicating with Web Services 185
7.7.3 “git”: Interacting with Git Servers 185

7.8 Special Modules (Control Flow Management, Etc.) 186
7.8.1 “meta”: Triggering Various Ansible Actions 186
7.8.2 “debug”: Generating Output during Playbook Runs 187
7.8.3 “fail”: Triggering a Failure 187
7.8.4 “pause”: Pausing and Optionally Reading Input 187
7.8.5 “wait_for”: Waiting for Certain Events 188
7.8.6 “wait_for_connection”: Waiting Until a Target Host Is Reachable 188
7.8.7 “assert”: Ensuring that Certain Conditions Are Met 189
7.8.8 “set_fact”: Setting Variables during Runtime 189

8 Modularization with Roles and Includes 191

8.1 Creating and Using Roles 191
8.1.1 The Concept of Roles in Ansible 191
8.1.2 A Simple Example of a Role 193
8.1.3 Using Roles in a Playbook 193
8.1.4 Plays with Roles and Tasks, “pre_tasks” and “post_tasks” 195
8.1.5 Dependencies between Roles 195
8.1.6 Choosing Other Startup Files 196
8.1.7 Creating New Roles with “ansible-galaxy” 197

8.2 The Online Ansible Galaxy Repository 198

8.3 Using Imports and Includes 198
8.3.1 “import_tasks” and “include_tasks” 198
8.3.2 “include_tasks” and Tags 199
8.3.3 Dynamic Loading of Variables with “include_vars” 201
8.3.4 “import_playbook” 201

8.4 Apache Once Again 202

8.5 Documentation (and Conventions) 206
8.5.1 "defaults/main.yml" as a Convention 207
8.5.2 "README.md" 208

8.6 Reusing Roles 209
8.6.1 Extending the Directory Structure 210
8.6.2 The Developer's Situation 212

9 Web Interfaces: AWX and More 213

9.1 Installation of Python Packages on Current Debian and Ubuntu Systems 213

9.2 Ansible Configuration Management Database ("ansible-cmdb") 214
9.2.1 Usage 215
9.2.2 Deploying the Page 215

9.3 Preparations for Operating More Demanding Applications 216

9.4 The Gitea Git Server 219
9.4.1 Commissioning and Logging In for the First Time 219
9.4.2 Checking In on Our Initial Project 220
9.4.3 Adding »README.md« and Useful Git Commands 222

9.5 AWX 223
9.5.1 Commissioning and First Login 223
9.5.2 Example Usage 224
9.5.3 Execution Environments 225
9.5.4 Conclusion 228

9.6 ARA 229
9.6.1 Test Setup 229
9.6.2 Further Possibilities 230

9.7 Other Applications Not Considered in Detail Here 230
9.7.1 Semaphore 231
9.7.2 Polemarch 231
9.7.3 Jenkins 232
9.7.4 Rundeck 232

9.8 Terminating or Deleting Applications That Are No Longer Needed 233

10 Additional Tools and Techniques 235

10.1 Ansible Vault ... 235
10.1.1 Before All Technology ... 235
10.1.2 Getting Started ... 236
10.1.3 Meaning of the Vault ID ... 238
10.1.4 Further Vault Commands ... 239
10.1.5 A Trick for Finding Variables Again ... 239
10.1.6 Encrypting Individual Variables ... 240
10.1.7 More Convenience or Automation ... 241
10.1.8 Convenience and Relative Security with a Password Client Script ... 242
10.1.9 Convenient and (Preferably) Secure with GNU Privacy Guard plus Pass ... 243

10.2 Debugging and Troubleshooting ... 244
10.2.1 Debug Mode and Verbosity Level ... 244
10.2.2 Improving the Readability of Outputs ... 245
10.2.3 Gathering Facts Takes Too Long ... 247
10.2.4 The Playbook Debugger ... 248
10.2.5 Static Code Analysis with “ansible-lint” ... 251
10.2.6 Check Mode and Diff Mode ... 254
10.2.7 Last but Not Least: The “debug” Module ... 256

10.3 Accelerating Playbooks with Pipelining ... 257

10.4 The Talking Cow ... 258

10.5 Ansible in Pull Mode ... 259
10.5.1 “ansible-pull”: Technique and Requirements ... 259
10.5.2 Getting Started ... 260
10.5.3 The Full Solution ... 261
10.5.4 What Might Still Be Missing? ... 263

11 Ansible and Docker 265

11.1 Installing Docker ... 265

11.2 Docker Modules ... 267
11.2.1 Preparations and Preliminary Considerations ... 267
11.2.2 A First Simple Example ... 268
11.2.3 Overview ... 269

11.3 An Example Application ... 274

11.4 Ansible and Docker Compose ... 278
11.4.1 Version Issues ... 281
11.4.2 Inline Specification with “definition” ... 281
11.5 The “docker” Connection Plug-in ... 282
11.6 Creating Images ... 283
11.6.1 Creating Images with “docker build” ... 283
11.6.2 “ansible-bender” ... 285
11.6.3 Creating Images with “ansible-bender” ... 286
11.6.4 Conclusion ... 288

12 Inventory Management: Advanced Methods 289

12.1 The “ansible-inventory” Command ... 289
12.2 Nested Groups ... 290
12.3 Static Inventories in YAML format ... 291
12.4 Creating on-the-Fly Inventories with “add_host” ... 293
12.5 Dynamic Groups with “group_by” ... 295
12.6 Dynamic and External Inventories ... 298
12.6.1 Inventory Scripts ... 298
12.6.2 Using Inventory Plug-ins ... 301

13 Ansible and the Cloud 303

13.1 Version Issues and virtualenv ... 304
13.2 Where to Store Keys, Tokens, Secrets, Etc. ... 304
13.3 Hetzner Cloud ... 305
13.3.1 Preparations on the Control Host ... 305
13.3.2 Preparations in the Cloud ... 306
13.3.3 Using Cloud Modules ... 306
13.3.4 Provisioning Cloud Servers ... 308
13.3.5 Inventorying Cloud Servers ... 309
13.3.6 Further Possibilities of the Inventory Plug-in ... 310
13.4 Amazon Web Services Elastic Compute Cloud ... 311
13.4.1 Preparations on the Control Host ... 311
13.4.2 Preparations in the Cloud ... 312

13.4.3 Using Cloud Modules ... 313
13.4.4 Provisioning Cloud Servers ... 314
13.4.5 Inventorying Cloud Servers ... 315
13.4.6 Further Possibilities of the Inventory Plug-in ... 316

13.5 Proxmox Virtual Environment ... 318
13.5.1 Preparations on the Control Host ... 318
13.5.2 Preparations in the Proxmox Virtual Environment ... 318
13.5.3 An Initial Test ... 319
13.5.4 Provisioning of Virtual Machines ... 320
13.5.5 Outlook ... 323

14 Ansible as an Orchestration Tool 325

14.1 Many Target Hosts for Testing ... 325

14.2 Altering the Sequence of Execution ... 326
14.2.1 “throttle” and “order” ... 328
14.2.2 “serial” ... 329
14.2.3 Faulty Hosts in “serial” Mode ... 330
14.2.4 Strategy Plug-ins ... 332

14.3 Delegation ... 335

15 Ansible and Windows 339

15.1 A Control Host Based on Windows ... 339
15.1.1 The Windows Subsystem for Linux ... 339
15.1.2 Cygwin ... 341

15.2 Windows Targets and Windows Remote Management ... 343

15.3 Preparations on the Control Host ... 343

15.4 Prerequisites on the Windows Side and Windows Remote Management Setup ... 344

15.5 Windows Remote Management Troubleshooting ... 345

15.6 Setup with an Active Directory and Kerberos ... 346

15.7 Windows Modules ... 348
15.7.1 “chocolatey.chocolatey.win_chocolatey”: Package Management with Chocolatey ... 349

15.7.2 “win_domain_user”: Managing Domain Accounts 349
15.7.3 “win_environment”: Managing Environment Variables 350
15.7.4 “win_regedit”: Managing Registry Entries 350
15.7.5 “win_updates”: Applying Updates 350

16 Callback Plug-ins 351

16.1 Stdout Callback Plug-ins 351
16.2 Aggregate and Notification Callback Plug-ins 353

17 Creating Your Own Collections and Modules 355

17.1 Namespaces, Names, and Setting Up a Collection Project 355
17.1.1 Ansible Project for Manually Testing the Collection 356
17.1.2 Where Is Work Currently Being Done? 357
17.2 Playbooks in Collections 357
17.3 Roles in Collections 358
17.4 Modules in Collections 359
17.4.1 Getting Started 359
17.4.2 Module Parameters 361
17.4.3 Modules with Python: Example Problem Statement 364
17.4.4 An Example Solution 365
17.4.5 Explanations and Further Possibilities 367
17.4.6 Embedded Documentation 369
17.4.7 Outlook 371
17.5 Plug-ins in Collections 371
17.5.1 An Example Callback Plug-in 371
17.5.2 Outlook 373
17.6 Deploying and Installing Collections 373

18 Developing and Testing with Molecule 377

18.1 Preparations and Setup 377
18.2 Getting Started 379

18.3 Develop ... 380
18.4 Testing with the Ansible Verifier ... 383
18.5 Testing with the Testinfra Verifier ... 385
18.6 The Complete Test Cycle ... 386
18.7 Outlook and Conclusion ... 387

19 Recipes, How-Tos, and Best Practices 389

19.1 New Projects ... 389
19.1.1 A Recommended “ansible.cfg” ... 389
19.1.2 A Template for a New Project ... 390
19.2 Administration ... 391
19.2.1 Building Simple Installers ... 391
19.2.2 Determining the IP Address of a Target Host ... 393
19.2.3 Managing “firewalld” ... 395
19.2.4 Applying Linux Software Updates ... 397
19.2.5 Initial Distribution of SSH keys ... 399
19.2.6 Passing Passwords on the Command Line ... 400
19.2.7 Ansible via a Gateway or Jump Host ... 401
19.3 Jinja Magic ... 401
19.3.1 Extending Maps or Lists during Runtime ... 401
19.3.2 Modifying and Combining the Elements of a List ... 403
19.3.3 Searching in a List of Maps ... 404
19.3.4 Filtering an Attribute from a List of Maps ... 405
19.3.5 Generating a Report from Target System Facts ... 406
19.3.6 Generating Passwords and Password Hashes ... 407
19.4 Tasks and Control Flow ... 408
19.4.1 Executing a Task Depending on a Previous Task ... 408
19.4.2 Executing a Task if the Host Is in a Specific Group ... 410
19.4.3 Avoiding Redundant Module Parameters with “module_defaults” ... 410
19.4.4 Dynamically Set Play Hosts ... 412
19.4.5 Reading Configuration Files ... 413
19.5 Miscellaneous ... 415
19.5.1 Simulating Functions ... 416
19.5.2 Managing Host-Specific Resources ... 417

20 What Could Be Better, and What Is Still Missing? 421

20.1 Tracking Long-Running Tasks ... 421
20.2 Finishing the Processing of a Role ... 422
20.3 Loops over Blocks ... 424
20.4 Locking with Concurrent Playbook Calls ... 425
20.5 Conclusion ... 426

Appendices 427

A Project-Specific Environment Variables with "direnv" ... 429
A.1 Installation and Shell Integration ... 429
A.2 Usage ... 430
A.3 Aliases and Functions ... 431
A.4 Command Execution Upon Entry ... 432

B The "pass" Password Manager ... 433
B.1 Installation and Preparations ... 433
B.2 Usage ... 434
B.3 Configuring GNU Privacy Guard Agent Cache Times ... 435

C Secure Shell ... 437
C.1 Server-Side Requirements for Using Secure Shell ... 437
C.2 Secure Shell Client Programs ... 438
C.3 Public Key Authentication ... 439
C.4 SSH Agents (Linux Client) ... 443
C.5 PuTTY, PuTTYgen, and Pageant ... 444
C.6 Windows Secure Copy ... 446
C.7 Advanced Configuration and Usage ... 447
C.8 "pssh" or "parallel-ssh" ... 452

D Regular Expressions ... 455
D.1 Motivation ... 455
D.2 Dialects ... 456
D.3 Basics ... 456
D.4 Regular Expressions: Extended Capabilities (Perl Compatible Regular Expressions) ... 458
D.5 Extended Capabilities (Portable Operating System Interface Extended Regular Expressions) ... 460

E **“vim” and “nano” Tips and Tricks** 463
E.1 “vim” 463
E.2 “nano” 464

The Author 467
Index 469
Service Pages I
Legal Notes II

Preface

It must have been 2014 when I first became aware of Ansible. At that time, I had no experience with configuration management systems; where necessary, I used shell scripts or Perl when more complex logic was required. So, I kept it in the back of my mind for a while, as old habits die hard.

Eventually, the time came for a new installation of a private server, and I took that as an opportunity to learn a bit more about this tool—via *learning by doing*, which is still the best method for me. And lo and behold: after developing about 240 tasks (individual steps), some of which were very tedious and distributed across several YAML files, I had the complete description in hand with which I could fully automate my server from the barebones delivery state to production-ready status. Fascinating! (I also enjoyed resetting the server during testing and watching the playbook run while sipping my coffee.)

So, alongside Perl (and Linux in general), I found a new favorite tool. When it was announced at the end of 2015 that Red Hat had acquired Ansible Inc., I was also quite sure that the topic would not be a flash in the pan. Since I'd always had a side job as an IT trainer, I began to develop and offer a training course on the topic of Ansible. (As trainers say, "If you want to learn more about a topic, then teach a course on it."). That also was very successful and educational. At the same time, I successfully utilized Ansible in several projects and thus gained additional experience with it.

When you discover something valuable, you naturally want to share it—and I wanted to publish something on the topic of Ansible, which led me to contact Rheinwerk Verlag, a publisher in Germany. The result was a chapter in the fifth edition of the popular *Linux-Server* handbook, which was published at the end of 2018. I had to fit the topic into only 80 pages due to space constraints, but in hindsight, that was enough for a start. (Writing books is more work than you think, and this one wasn't even my own.) By the end of 2019, we all thought it would be a good idea to dedicate a book to the topic of Ansible—and in September 2020, the first edition actually appeared and I was ready for a vacation.

Of course, time hasn't stood still since then—especially not in IT. That's why you are now holding an updated and expanded English edition in your hands. What's new since it first published in German? I have "dusted off" many areas, reassessed things, and incorporated new experiences from my practice. A completely new topic I added is the *Proxmox* virtualization platform and its management via Ansible. Additionally, I have become very fond of the Python programming language over the past few years, so the example scripts are now consistently implemented in Python. (Goodbye Perl! I loved our time together!)

What remains is definitely fun with Ansible—an extremely successful tool. I know of no other software that does so many useful things while annoying me so rarely.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

I would like to thank the many people in the Ansible community who have contributed to the development, improvement, and dissemination of this wonderful software—without whom, this book would never have existed!

Last but not least, my thanks go to my partner Ana and our daughter Lena, who have always shown great understanding and given me the freedom I needed to work on this book.

Now, all that remains is for me to wish you much fun reading and hope that this book will be helpful to you in your daily work or your personal projects.

Axel Miesen
Tauberbischofsheim, Germany
April 2025

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

About This Book

Now, I would like to take a moment to explain the thought process behind the various formatting options. I'll also give you an overview of the software and the Linux versions featured in the book.

Formatting

The formatting used in this book is broken down as follows:

- **Console/command line**
  Console commands and their outputs are marked by the use of a monospace font. The inputs you need to make are shown in bold, while any outputs are in normal font:

  ```
  $ ls -l /usr/local/bin
  total 177640
  -rwxr-xr-x 1 root root 51454104 Jul 15 13:38 kubectl
  -rwxr-xr-x 1 root root 11636736 May 27 20:05 lazydocker
  […]
  ```

- **Normal and privileged rights**
  Unless I specifically point it out in the text, the prompt symbol will let you know whether to perform the input with normal user rights or with root privileges. A dollar sign indicates you should perform the input with normal user rights:

  ```
  $ whoami
  ```

  And a number sign indicates you should perform the input with root privileges:

  ```
  # useradd -m user2
  ```

  Alternatively, especially when it comes to individual commands, I sometimes put a `sudo` in front:

  ```
  $ sudo useradd -m user2
  ```

  Since `sudo` is probably the de facto standard for privilege escalation in the Linux world, such an example should be easy for you to understand. If `sudo` is not available on your specific system, then just know it's a hint that you need root privileges for the command.

- **Entering long commands**
  Some of the command lines presented in the text extend over several printed lines, although they are usually entered on one line in the computer. Basically, you could

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

enter these commands exactly as they are printed; you can use the usual »\« shell character for line continuation, or you can take advantage of the fact that the shell has no problem with line breaks at certain symbols like »|« and »&&«:

```
$ curl -sSL https://github.com/direnv/direnv/releases/download/\
v2.34.0/direnv.linux-amd64 \
    -o /usr/local/bin/direnv

$ history    | awk '{print $2}' |
    sort     | uniq -c |
    sort -rn | head
```

- **Path specifications, programs, options, and package names**
  Path specifications, programs, options, and package names are usually highlighted font styles like italics or inline code characters, as in the following examples:
  - The */etc/passwd* file is essential.
  - You can search for it with `grep`.
  - The `--color=auto` option is useful for this.
  - You should also install the *tree* package.
- **Internet references**
  Even in a book like this, it sometimes isn't practical or sensible to go into great detail. For this reason, I have referred to internet addresses in appropriate places, and I've highlighted the URLs in italics, like this: *https://www.ansible.com*.
- **Materials for the book**
  You don't have to painstakingly type out the example files or listings in this book. Instead, you'll find various materials or files related to this book on GitHub at *https://github.com/ansible-buch*. An alternative resource you can use (because GitHub might not be freely accessible forever) is the publisher's website: *https://www.rheinwerk-computing.com/6068*. There, you can scroll down to the **Product Supplements** box and click **Supplements list**, where you'll see a folder of downloadable files along with a brief description of them.

Additionally, there are highlighted boxes in this book that contain information that is useful but outside the context of the current topic. We have given the following titles to the boxes to help you identify the type of information contained in each box.

Note

Boxes with this title contain information about additional topics or important content that you should note.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Tip

Boxes with this title refers to specifics items or ideas that you should consider, and they also warn you about frequent errors or problems that can occur.

Warning

Boxes with this title warn you of important points that are potentially confusing or likely to be overlooked.

If you have any suggestions for improvement, I would, of course, welcome your feedback.

Ansible Versions

The topics we cover in this book work with Ansible version 2.10 (released in September 2020) or later. Most of them also work with version 2.9 or possibly with an even older version, but if you are interested in the topic of *collections* (and you should be!), you really need version 2.10 or, even better, 2.11. The examples shown in the book were mostly tested with version 2.14 or Community Package Version 7 under Debian 12.

If appropriate or necessary, I will briefly address differences or incompatibilities with previous versions.

Platforms and Linux Distributions

This book considers the following Linux distributions:

- Debian 12
- Rocky Linux 9
- openSUSE Leap 15
- Ubuntu 24.04 LTS

Why these in particular? From a technological perspective, they should cover everything you might encounter in a professional environment. (Rocky Linux is very close to Red Hat Enterprise Linux (RHEL), and openSUSE Leap is very close to SUSE Linux Enterprise Server [SLES].) Furthermore, all these distributions are completely free, so there are no financial barriers to their use.

The examples and approaches shown in this book will definitely work with other Linux distributions as well, but within the scope of a book, everything shown should be as well tested as possible. Testing just these few distributions took a considerable amount of time, so please understand if your favorite distribution is not on the list.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 1
Introduction and Installation

In this chapter, I would like to explain what Ansible is, and also what it is not. After that, I will show you a way to quickly set up a test environment so that we can then proceed to Ansible installation.

1.1 What Is Ansible?

Ansible is a tool for solving problems that mostly arise on the "Ops" side of the DevOps world. Anyone who has ever been responsible for a small number of servers can confirm that a fully manual approach to administration or configuration is at least highly error-prone, time-consuming, and difficult to follow—and anyone who has ever been responsible for a large number of servers will confirm that using such an approach with a large number of servers is practically impossible.

Perhaps you've already worked with scripts (e.g., Shell, Python, Perl) in such a situation, or perhaps you're familiar with an alternative product like Puppet, Chef, SaltStack, or CFEngine. In either case, I am pleased that you now want to engage with Ansible, which offers you, among other things, the following possibilities:

- Automation of system provisioning
- Automation of software deployment
- Configuration management

Ansible is indeed an extremely flexible tool, but I will spare you the all-too-often overused comparison with the proverbial Swiss cutting tools. You can use Ansible to deploy software to a single machine, or you can use it to manage configurations on hundreds of servers. Moreover, it is (relatively) easy to learn, and most users even find it enjoyable!

1.1.1 Basic Features

Ansible is characterized by the following features:

- Ansible is programmed in Python. Originally, it was developed in Python 2, but since Ansible version 2.5, Python 3 has been fully supported.
- Ansible primarily manages its hosts via Secure Shell (SSH) and only requires Python to be installed on the target systems (along with some additional Python modules in

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

special cases). In particular, Ansible is *agentless* because it does not require any special services on the target systems.

- Ansible mainly uses YAML as a configuration language.
- Thousands of modules are available for Ansible for various management tasks.
- Ansible offers quick wins but also scales well in complex scenarios.
- Ansible is very well documented (see *https://docs.ansible.com*).

1.1.2 Core Components

Typically, an Ansible project includes the following components:

1. The *inventory* is an index of machines that are to be managed with Ansible, and it usually needs to be created by the user.
2. The *playbooks* are user-created procedures that describe in individual steps (*tasks*) how to achieve various configuration goals.
3. The *modules* are functional units provided by Ansible that perform the actual work. Each step in a playbook is ultimately nothing more than a call to a module.
4. *Roles* are used to structure larger projects in a modular, maintainable, and reusable way. They have at least one thing in common with classes from object-oriented programming: at first, you don't need them, and later, you can't do without them.

What Does Ansible Mean?

The name *Ansible* comes from science fiction. An *ansible* is a device for faster-than-light communication. It was originally conceived by Ursula K. Le Guin, and many other authors have adapted the idea. Specifically, Ansible's creator Michael DeHaan borrowed the term from the book *Ender's Game* by Orson Scott Card, in which an ansible is used to communicate with fleets and bases over very large distances.

1.1.3 Declarative or Imperative?

Configuration management tools like Ansible are often categorized as declarative or imperative, which in turn describes the character of the domain-specific language (DSL) used in the specific product.

In *declarative* languages, you describe the desired end state, whereas in *imperative* languages, you describe the *how*—that is, the path to get there. The SQL database language is clearly imperative (e.g., `CREATE TABLE`, `INSERT INTO` …), while the XML-based SVG graphics format is a nice example of a declarative language, as illustrated in the code snippet below:

```
<svg xmlns="http://www.w3.org/2000/svg" height="480" width="680">
  <circle cx="340" cy="240" r="100" />
</svg>
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Here, a circle is simply "wished into existence," and someone has to take care of getting the job done.

Into which category does our Ansible fall? It is not clearly definable; we are dealing with a kind of hybrid. There are more declarative examples like this:

```
- service:
    name: httpd
    state: started
```

And there are also more imperative-looking calls:

```
- copy:
    src: test.txt
    dest: /tmp
```

Ultimately, it is a moot point and often a matter of interpretation. You could read the second example as "Copy the *test.txt* file to the */tmp* directory," or as "I want a copy of the *test.txt* file in the */tmp* directory."

If you ever have nothing better to do, try googling "Ansible declarative imperative," where you'll find an infinite number of opinions on this topic. You're better off just forgetting about it—and if you can't, you can philosophize about it at the DevOps roundtable.

1.2 What Is Ansible Not?

Just to be safe, so you don't approach this tool with false expectations, here are some notes on what Ansible is not:

- Ansible is not a flashy GUI-based tool. At its core, it consists of various command line tools along with their input files, and you need to understand and master all of them. There are indeed GUIs (graphical applications) for Ansible, but if you don't understand Ansible's internal workings, you won't get much out of them.
- Ansible does not have an excessively high level of abstraction. You cannot tell Ansible something like "Ensure the operation of a web server." Rather, you must give it step-by-step instructions, like "Install the `httpd` package, start the `httpd` service and integrate it into the autostart, open ports 80 and 443 via `firewalld`, etc."

  So, here's a simple rule of thumb: *If you don't know how to do it without Ansible, then Ansible can't help you do it either.*
- Ansible is not a pure admin tool. It's a tool from the DevOps world, so you must be willing to engage with concepts from the development world—such as if-statements, loops, error handling, access to (complex) data structures, etc.
- Nevertheless, Ansible is *not a programming language* and is not intended to be one by design. (However, *if* you have some programming knowledge, it will certainly help you!)

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

1.3 History and Versions

In 2012, Michael DeHaan introduced the first version of Ansible. The company he founded, *Ansible Inc.,* was acquired by Red Hat at the end of 2015—and since then, the ongoing development of Ansible has been significantly influenced by Red Hat. This has been very beneficial for the product so far.

1.3.1 Versions

Ansible's versioning scheme and especially the internal organization of its further development were significantly changed by Ansible version 2.10 and subsequent versions. Since 2.10, the Ansible core and additional components like modules and plugins have been developed independently. The Ansible core continues to be developed with the old versioning scheme and is now called *Ansible Core* (in version 2.11 and later versions; in version 2.10, it was briefly called *Ansible-base*). The numerous additional components that are available are now called *collections*, and each one follows an individual versioning scheme.

For most end users, the most interesting thing now is the *Ansible community package*, which started with version 3.0. This is essentially just a meta-distribution that determines the version of the included Ansible Core and the selection and versions of the bundled collections. The version numbers now also follow the principle of semantic versioning (`Major.Minor.Patch`). Here's an overview of some of them:

- **Ansible 2.8 and older**
  This is the classic complete package (for us, it's the "Stone Age").
- **Ansible 2.9**
  This is the classic complete package, with collections as a tech preview.
- **Ansible 2.10**
  This represents the end of preparations and the division into Ansible-base and collections.
- **Ansible 3.X**
  This version introduced the community package, which consisted of Ansible-base 2.10 plus collections.
- **Ansible 4.X**
  This version includes a community package that consists of Ansible Core 2.11 plus collections.
- **Ansible 5.X**
  This version includes a community package that consists of Ansible Core 2.12 plus collections.
- **...**

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

- **Ansible 11.X**
  The latest version is Ansible 11.X, which contains a community package that consists of Ansible Core 2.18 plus collections. Subsequent versions of Ansible will continue to have whole-number designations and Ansible Core designations of 2.X.

You can find the exact status current releases at *https://docs. ansible.com/ansible/latest/reference_appendices/release_and_maintenance.html*.

1.3.2 Supported Python Versions

There are Ansible requirements related to the Python version you use, and you also have to distinguish between the control host and the target host. On the website, there is the Ansible Core support matrix, where all compatible versions are presented clearly and comprehensively. Some conclusions from this are as follows:

- For the control host, Python 2 support ends with Ansible Core version 2.11.
- For target hosts, Python 2 support ends with Ansible Core version 2.16.
- If you operate a control cost with a standard Rocky 9 Linux, the latest Ansible version you will be able to install there is Ansible Core 2.15 because the supplied Python version (3.9) is the limiting factor in this configuration.
- If you are running target hosts with a standard openSUSE Leap, your control host must run with Ansible Core version 2.16 at latest because openSUSE Leap ships with Python 3.6 by default. (However, there is also a *python311* package that you could install on the SUSE targets if needed.)

Note

Please don't worry about versions too much at the beginning. In all current Linux enterprise distributions, the versions of the Ansible community and Python packages are chosen so that you'll rarely encounter any compatibility issues—which means you can safely focus your attention on more interesting things!

1.4 Setup/Lab Environment

You've purchased this book to familiarize yourself with Ansible, and the tried-and-true way to do this is learning by doing (i.e., you should follow the examples and procedures that are relevant to you on the job). Ideally, you should have a test or lab environment available, and I would like to describe that now.

First of all, you need some kind of UNIX-like host on which the Ansible software will be installed (the so-called *control host* or *control machine*). The software will work on any

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

common, reasonably up-to-date Linux distribution, and it will also work on macOS, Solaris, etc. There is no native Windows version of Ansible; however, through workarounds (e.g., Cygwin or WSL), you can make it run on Windows. However, please refrain from doing so at this point because too many tricks and tweaks would be necessary.

Additionally, you need at least one more host that should you should administer via Ansible (a so-called *target host* or *managed node*). In the beginning, we'll assume you're using Linux systems here—and much later, you'll see that you can manage Windows systems, network devices, and cloud environments by using Ansible. The most important requirement at this point is that the control host can reach its target hosts (in the case of Linux) via SSH.

Of course, I have no way to know what your work or test environment looks like. Perhaps your workstation has long been a Linux system and you rented three VServers in the cloud last week—and if so, you can start right away!

Or not . . . meaning you may need help choosing a test environment. In that case, the main test environment that I suggest you use while working through this book is shown in Figure 1.1. I'll also show you how to set up such an environment if you don't want to build one on your own.

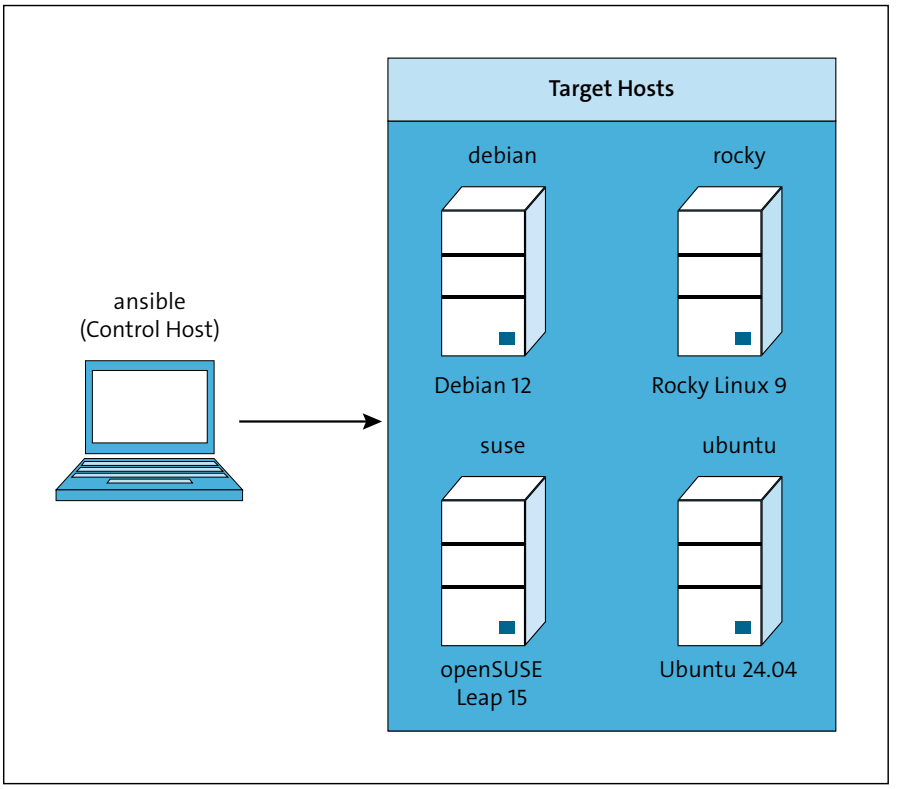


Figure 1.1 Our Example Setup

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The *Vagrant* software is excellent for providing a virtual test environment. It only requires a standard virtualization solution (e.g., VirtualBox, VMware, Hyper-V), and it can then provision a defined set of virtual machines by using a description file (the *Vagrantfile*). At this point, I'll assume that you'll use VirtualBox as the virtualizer, and you can get the required software here:

- Vagrant (*https://www.vagrantup.com*)
- VirtualBox (*https://www.virtualbox.org*)

Once you've installed these two virtualization tools on your system, we can turn to creating the virtual test systems. Listing 1.1 shows a Vagrantfile, which you should simply place in any new directory as follows:

```
#
# Please adjust if necessary:
#
NETWORK_PREFIX = "192.168.150"

Vagrant.configure("2") do |config|

  # Set default RAM size for all boxes:
  config.vm.provider "virtualbox" do |vb|
    vb.customize ["modifyvm": :id, "--memory": "1024"]
  end

  config.vm.define "ansible" do |ansible|
    ansible.vm.box = "bento/debian-12"
    ansible.vm.hostname = "ansible"
    ansible.vm.network :private_network, ip: "#{NETWORK_PREFIX}.100"
  end

  config.vm.define "debian" do |debian|
    debian.vm.box = "bento/debian-12"
    debian.vm.hostname = "debian"
    debian.vm.network :private_network, ip: "#{NETWORK_PREFIX}.10"
  end

  config.vm.define "rocky" do |rocky|
    rocky.vm.box = "bento/rockylinux-9"
    rocky.vm.hostname = "rocky"
    rocky.vm.network :private_network, ip: "#{NETWORK_PREFIX}.20"
  end

  config.vm.define "suse" do |suse|
    suse.vm.box = "bento/opensuse-leap-15"
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    suse.vm.hostname = "suse"
    suse.vm.network :private_network, ip: "#{NETWORK_PREFIX}.30"
  end

  config.vm.define "ubuntu" do |ubuntu|
    ubuntu.vm.box = "bento/ubuntu-24.04"
    ubuntu.vm.hostname = "ubuntu"
    ubuntu.vm.network :private_network, ip: "#{NETWORK_PREFIX}.40"
  end

end
```

Listing 1.1 Vagrantfile, Provisioning Test Environment

This file defines the five machines shown in Figure 1.1. However, you need to do a little preparation for your environment first by following these steps:

1. Create a new *host-only network* in the VirtualBox application.

   To do this, you navigate to your **Settings,** switch to **Expert**, and click **File • Tools • Network Manager**. There, you click on **Create** and then note the IPv4 address or adjust it according to your own preferences.

   In the book, I use this as an example:

   `IPv4 address:` **`192.168.150.1/24`**

   Unless you have some reason not to, it makes sense for you to follow this example. Otherwise, you may need to adjust a few example files later, which should not be a problem.

   Next, please set the DHCP server of this new network to `disabled`:

   `DHCP Server:` **`disabled`**

   We don't need it, and we also don't want to run the risk that it could somehow interfere:

   If you are running VirtualBox on Linux, then the default setting does not allow arbitrary choice of host-only network addresses. To allow arbitrary networks on Linux, you must first allow this as a root by creating an */etc/vbox/networks.conf* file with the following content:

   ```
   * 0.0.0.0/0 ::/0
   ```

   Listing 1.2 "/etc/vbox/networks.conf": Allow Arbitrary Networks on Linux

   If necessary, you may need to close and reopen the VirtualBox app after completing this step.

2. Now, you adjust the `NETWORK_PREFIX` in the Vagrantfile (if necessary) with your VirtualBox configuration.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The virtual machines are provisioned with 1 GB of RAM each. In total, our setup initially requires a maximum of 5 GB of RAM, but in reality, it will likely measure somewhat less. If your host system does not provide this amount of main memory, you'll have to simply omit one target host or another (which would be a shame, as later examples work with all these hosts).

After you complete this preparation, you'll be able to bring the entire scenario online. To do this, you switch to the directory in the command line environment where you have placed the Vagrantfile and enter the following:

```
$ vagrant up
```

After some waiting time, all machines should be running and `ping`-able according to their configured IP addresses:

```
$ ping 192.168.150.100
$ ping 192.168.150.10
$ ping 192.168.150.20
[…]
```

Then, for example, you should be able to log in to the control host; if you are working on Windows, you are welcome to use familiar software like *PuTTY* for this purpose. Your username and password will both be `vagrant`, and the Vagrant software itself also offers a convenient way to log in:

```
$ vagrant ssh ansible
vagrant@ansible:~$ exit
```

Note

The `vagrant` user account has unrestricted `sudo` privileges by default. So, if you need root privileges, you can simply prepend `sudo` to your command or (if necessary) obtain a permanent root shell by using `sudo -i`.

Before we leave the topic of Vagrant, you should take a look at Table 1.1 for an overview of the most important Vagrant commands:

Commands	Effects
`vagrant global-status`	It gives you an overview of what is running and what is not.
`vagrant box list`	It lists downloaded images.
`vagrant box update`	It updates downloaded images.

Table 1.1 Most Important Vagrant Commands

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Commands	Effects
`vagrant up [<NAME>…]`	It starts the machine(s).
`vagrant halt [<NAME>…]`	It stops the machine(s).
`vagrant reload [<NAME>…]`	It restarts the machine(s).
`vagrant destroy [<NAME>…]`	It discards the machine(s).
`vagrant snapshot push [<NAME>…]`	It snapshots the machine(s).
`vagrant snapshot pop [<NAME>…]`	It returns to a snapshot
`vagrant snapshot list`	It lists all snapshots.

Table 1.1 Most Important Vagrant Commands (Cont.)

Machine-related commands like `up`, `halt`, etc., must always be called in the directory or below the directory that contains the corresponding Vagrantfile!

1.5 Ansible Installation on the Control Host

An Ansible community package is available in the main package sources of all major distributions—but with Rocky and RHEL, the situation somewhat different. There, one usually resorts to Extra Packages for Enterprise Linux (EPEL, see below). Table 1.2 shows the state of affairs as of the fall of 2024.

Distribution	Packaged Version
Debian 12	7.7.0
Debian 11	2.10.8
openSUSE Leap 15.6	9.4.0
openSUSE Leap 15.5	2.9.27
Ubuntu 24.04 LTS	9.2.0
Ubuntu 22.04 LTS	2.10.8
Rocky Linux 9	Ansible Core 2.14.14
Rocky Linux 8	–

Table 1.2 Ansible Versions in Different Distributions

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Note

For clarity's sake, I would like to reiterate that since Ansible is *agentless*, you only need to install the Ansible software on the control host—which, in our lab environment, is a Debian 12 system. You currently need nothing except SSH and Python on the target systems.

You can generally install distribution packages quite quickly by inputting the following code for each package:

- **Debian 12** (in our lab environment, this is our control host) **or Debian 11**

```
# apt update
# apt install ansible
```

- **Rocky Linux 9**

  As shown in Table 1.2, the standard Rocky Linux 9 package only includes the Ansible Core package. As soon as you need modules or collections that are not included in this base package, you would have to install them directly from the internet (see Chapter 7, Section 7.1). In many environments, using the Ansible package from the EPEL supplementary repository might be easier, which is why I recommend that you start with this approach :

  First, activate the EPEL supplementary repository:

```
# dnf install epel-release
# dnf install ansible
```

  Then, for simplicity's sake, proceed in exactly the same way under Rocky Linux 8.

- **openSUSE Leap 15.6**

```
# zypper install ansible
```

  If you are running an older openSUSE version, the packaged Ansible version might be too old. In that case, you would need to resort to an installation with PIP (see below).

- **Ubuntu 24.04 LTS, Ubuntu 22.04 LTS**

```
$ sudo apt update
$ sudo apt install ansible
```

Once the installation is successful, go ahead and perform a test call:

```
$ ansible --version
ansible [core 2.14.16]
  config file = None
  […]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

You should also make entries in */etc/hosts* for all involved hosts on the control host, because even in the test lab, you want to avoid working with IP addresses as much as possible. If DNS-based name resolution is available to you in reality, then you can, of course, skip this step:

```
#
# Please adjust according to your environment:
#
192.168.150.10  debian.example.org  debian
192.168.150.20  rocky.example.org   rocky
192.168.150.30  suse.example.org    suse
192.168.150.40  ubuntu.example.org  ubuntu
```

Listing 1.3 Example Excerpt from "/etc/hosts"

1.6 Installation via PIP (plus virtualenv)

Note

If the Ansible package from your distribution or your additional repository is sufficiently up-to-date and you are not currently interested in Python's virtualenv feature, then you can skip this section and continue on to Section 1.7.

The second installation option is via Python's package management. You might consider this option in the following cases:

- Your distribution provides only outdated packages or no suitable packages at all.
- Using additional repositories is not an option.
- You would like to use different Ansible versions in parallel.

First, you need to ensure the necessary prerequisites as root. You require two distribution packages, which will automatically pull all other dependencies:

```
$ sudo apt install python3-pip python3-venv
```

With SUSE and Rocky, you only need *python3-pip* because *venv* is a standard module in Python 3. However, a normal Debian Python installation is somewhat leaner; the *python3-full* package would be the all-inclusive solution here.

At this point, the fastest way to install Ansible is direct installation via Python package management, for which you would only need the *python3-pip* package. You can use the following as root, if you really want to:

```
$ sudo pip3 install ansible
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

However, this is no longer recommended, and a current Debian family system would even need to be persuaded with the `--break-system-packages` option.

A very good alternative is to perform the installation in a virtualenv environment. This method has the advantage that you can easily work as an unprivileged user and could even run multiple versions in parallel. This method also avoids the potential problem of PIP making changes to the system that are difficult to undo.

So, feel free to enter the following commands as an unprivileged user:

```
Set up a virtual environment (you can freely choose the directory):
$ python3 -m venv ~/venv/ansible

Enter the environment:
$ source ~/venv/ansible/bin/activate

Update PIP if necessary during the *first use* of the environment.
You do this in case serious problems arise later:
$ pip3 install --upgrade pip


Perform ansible installation (without "==VERSION" you will get the latest ver-
sion possible for your Python):
$ pip3 install ansible==9.2.0
[…]

$ ansible --version
ansible [core 2.16.8]
[…]

Leave the environment:
$ deactivate
```

If you wish, you can now manage as many different Virtualenv environments as you want with as many different Ansible versions (or any other Python software packages) as you like.

1.7 Authentication and Authorization on Target Hosts

Now, let's move on to the target hosts. We want to perform administrative tasks on them with Ansible, so you need to clarify some questions for each target host (for now, we are talking about Linux hosts):

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

- **Which user account can be accessed via SSH from outside, and which authentication method should I use?**
  The most common method is public key authentication, but sometimes, only traditional passwords are available.

  Less common, but not unusual, is SSH Kerberos authentication. Setting this up is a completely independent and highly complex topic that we cannot cover here. However, if you have a fully set up environment in front of you, no further configuration is required from the perspective of the Ansible command line—Ansible can use a valid Kerberos ticket just like a human user can.
- **Which other system access methods (e.g., su or sudo) should I use?**
  Once Ansible has successfully authenticated, you need to determine whether root privileges are already available or whether an identity change or privilege escalation is still required?

The current state of our lab environment is very uniform: We access all systems via the `vagrant` account with the `vagrant` password, and we perform privilege escalation with `sudo`.

Now, let's walk through an initial test with UNIX password authentication.

Note

We will consistently work with SSH keys in the lab. If this is also your preferred authentication method or passwords do not matter in your environment, then feel free skip to the next section.

You probably know that automation over SSH with passwords is rather difficult, so for the moment, let's just assume that we don't want to change the authentication method. With this in mind, you need to install the *sshpass* package on the control host:

```
This is only necessary if Ansible is going to authenticate with passwords:
$ sudo apt install sshpass
```

Once you've installed the package, you can test your target hosts with regard to Ansible. However, the prerequisite for this is that our control host must know the host keys of all target hosts. In the lab, you can accomplish this very easily with the `ssh-keyscan` command:

```
$ mkdir -p ~/.ssh && chmod 700 ~/.ssh
$ ssh-keyscan -t rsa debian rocky suse ubuntu > ~/.ssh/known_hosts
```

Below, you'll see two test calls in which you'll be prompted to enter the login password each time. Your goal is to see a green `pong` in response to each Ansible `ping`—which ensures, among other things, that a Python interpreter is available on the other side. For now, you can ignore all warning messages:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
$ ansible all -i debian,rocky,suse,ubuntu -u vagrant -m ping -k
[…]
This is a single target system; note the comma!
$ ansible all -i debian, -u vagrant -m ping -k
[…]
```

We will clarify the semantics of the `-i` option later (note the comma after a single hostname). The `-u` option determines the login user, and the `-k` option prompts for the login password. With the additional `-b` option, you can initiate an identity change to the root user after login. The default method for this is `sudo`, unless you choose another method with `--become-method`. If an additional password entry is required, you must also specify the additional option with `-K`.

Here are some commands that might be suitable for other environments:

```
$ ansible all -i somecentos, -m ping -u ansible -k -b --become-method su -K
$ ansible all -i someubuntu, -m ping -u ansible -k -b -K
```

1.8 Setting Up SSH Public Key Authentication

To operate Ansible, you typically set up SSH public key authentication from the control host to the target hosts. (And please do this in our lab.) We use the configured `vagrant` user as the starting account on our control host. (Working permanently as root is neither necessary nor advisable here.)

I assume you know how to set up public key authentication—but if not, here is a quick-and-dirty approach you can use in the test environment:

```
vagrant@ansible:~$ ssh-keygen
(… accept all defaults …)

Do this to distribute the key:
vagrant@ansible:~$ ssh-copy-id vagrant@debian
vagrant@ansible:~$ ssh-copy-id vagrant@rocky
vagrant@ansible:~$ ssh-copy-id vagrant@suse
vagrant@ansible:~$ ssh-copy-id vagrant@ubuntu
```

If you require a bit more background knowledge about SSH for your environment, you can read Appendix C, where you will find further information on secure private keys, SSH agents, and much more.

Note

Of course, you can also generate or use a key with a filename that differs from the default (e.g., *~/.ssh/ansible_key*). With `ssh-copy-id`, you must then specify the key by using `-i`.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

1.9 An Ad Hoc Test without Any Configuration

Now, you can (and should!) use an Ansible `ping` call to test whether the Ansible working environment is set up correctly. Before you begin, here are a couple of tips:

- If the filename of your private SSH key differs from the default, please add `--private-key <PATH>` accordingly.
- Specifying `-u <USERNAME>` is actually unnecessary, as the target account is the same as the source account.

Here's what you do to set up the `ping` call:

```
$ ansible all -i debian,rocky,suse,ubuntu -u vagrant -m ping
debian | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
rocky | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
[WARNING]: Platform Linux on host suse is using the discovered
Python interpreter at /usr/bin/python3.6, but future installation
of another Python interpreter could change the meaning of that path.
See https://docs.ansible.com/[…] for more information.
suse | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.6"
    },
    "changed": false,
    "ping": "pong"
}
ubuntu | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Despite any warning or informational messages you receive, you should find that all target systems respond with a friendly pong. Since version 2.8, Ansible has become significantly smarter in terms of detecting the Python interpreter on the other side. However, it now also shares its wisdom with the world through extensive warning and informational messages; we will soon make this a bit quieter with configuration.

If you receive serious error messages at this point, then it's usually due to one of two reasons: the Python interpreter is missing or your Ansible is too old and does not correctly recognize the existing Python version. In the first case, you should equip your target systems with a Python installation, and in the second case, you should run a sufficiently current version of Ansible.

1.10 Another Note on Migrating from Older Versions

Ansible has a rather eventful development history. Especially (but not only) with the jump to version 2.0, there were many earthshattering changes—but since version 2.4, it's been much quieter.

If you want to migrate older Ansible projects to a newer version but you encounter difficulties, you might want to take a look at the Porting Guides at *https://docs.ansible.com/ansible/devel/porting_guides/porting_guides.html*.

However, since only the changes between consecutive versions are described there, it may not be very helpful if your playbook from the 2.4 era no longer runs under Ansible 2.17. The most common cause of such problems is usually incompatibilities between older and newer versions of individual modules, so you should carefully examine the warning or error messages and consult the current module help page. As long as you have enough relevant knowledge and experience, this will help lead you to the goal more quickly.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 2
Basic Setup and Initial Inventory Management

In this chapter, you will find suggestions on how to structure your work at the directory level for Ansible. Additionally, we will deal with the *inventory* (which is essentially the index of target hosts) that you need for almost every scenario in Ansible.

2.1 Setting Up the Directory Structure

Ansible gives us few fixed guidelines when it comes to which structures we must create at the directory and file levels. The following two guidelines make sense to me:

- You should locate each Ansible project within a single folder. The advantage of this is that you can easily back up this folder at any time or commit it to a version control system without having to worry about forgetting something important.
- You should work on the control host as an unprivileged user (as you already did when you set up SSH public key authentication).

Due to the second point, you should rule out using the */etc/ansible* folder (which was at least previously created by default during installation from distribution packages). Just ignore it if it still exists on your system.

Every typical Ansible project needs a configuration, an inventory, and one or more playbooks—so let's start generically with a project directory named *~/ansible/projects/start*, which will soon contain all these things:

```
$ mkdir -p ~/ansible/projects/start && cd $_
```

I like to use the Bash special `$_` variable often because it contains the last argument of the last command.

But do you really need such a deep directory hierarchy right away? At the beginning, not really; a directory like *hello* or *ansible-test* would suffice. However, with the proposed structure, you'll be somewhat prepared for the future. Eventually, you'll want to manage multiple projects in parallel, and you'll not only work on projects but also develop roles, modules, plug-ins, and collections.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

2.2 Basic Configuration (ansible.cfg)

Since we want to operate Ansible with some settings that deviate from the default, we need a configuration file. You can search for this file in the following locations (in order; the first one wins):

1. The content of the `ANSIBLE_CONFIG` environment variable
2. *ansible.cfg* in the current directory
3. *~/.ansible.cfg*
4. */etc/ansible/ansible.cfg*

According to our specifications, we recommend option 2. You should place the following first version of a configuration file in your *~/ansible/projects/start/* project directory:

```
[defaults]
inventory = ./inventory
interpreter_python = auto_silent
host_key_checking = False
```

Listing 2.1 First Version of Configuration File: "ansible.cfg"

Here, Ansible is configured using a variant of the INI format. I need to briefly mention the following non-obvious peculiarities here:

- Comments begin with a number sign (#) or a semicolon (;). You can only use a number sign at the beginning of a line, whereas you can use a semicolon anywhere.
- If a line begins with one or more white spaces, it is considered a continuation of the previous line. As a result, *one* space is used as a separator.

Now, let's move on to the explanations of the actual configuration settings.

- `inventory`
  With the `inventory` directive, you set the path to an (as yet to be created) inventory file. The default here would have been */etc/ansible/hosts*, which is unsuitable for us.
- `interpreter_python`
  With the `interpreter_python` directive, you can either set the path to the Python interpreter in the target hosts or choose one of four additional settings that allow you to control how Ansible looks for an appropriate Python interpreter. The `auto` default is fine in most cases, and the `auto_silent` variant does the same but spares us the constant notification messages regarding automatic interpreter detection.

  If you unexpectedly encounter issues with the Python version you want to use, please refer to the official documentation at *https://docs.ansible.com/ansible/latest/reference_appendices/interpreter_discovery.html*.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

- `host_key_checking`
  By setting `host_key_checking = False`, you can disable the checking of SSH host keys, which you may want to do because lab or test hosts are often newly added, reprovisioned, or installed. In our test environment, we can certainly encounter events like this one:

  ```
  The authenticity of host '[…]' can't be established.
  ECDSA key fingerprint is 8a:a3:a5:43:c6:e4:6c:11:3a:c9:2b:97:94:23:40:c9.
  Are you sure you want to continue connecting (yes/no)?
  ```

  And here's another one:

  ```
  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  @    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
  ```

  You can safely do without host key checking; in production operation, I strongly advise against this setting (or you should only use it there if you are fully aware of the implications)!

At this point, you may be asking whether the configuration file has also been found. As long as you are in the project's base folder, Ansible should find and use the new configuration file if you input the following:

```
$ ansible --version
ansible [core 2.14.16]
  config file = /home/vagrant/ansible/projects/start/ansible.cfg
  […]
```

However, as soon as you change your current working directory, Ansible will fall back to another (or no) configuration, which is not in your best interests. But since it is not always convenient to make every call in the base folder, you should always refer to our configuration file with the `ANSIBLE_CONFIG` environment variable (which is option 1 from the list at the beginning of this section).

I strongly encourage you to solve this problem with the *direnv* software. Of course, you could set this environment variable in your *.bashrc*, for example, but sooner or later, you will manage multiple Ansible projects, each with its own configuration, and then, it will become difficult again.

So, please take a short detour to Appendix A, where you will learn how to install, initialize, and use direnv in general. After that, you can return here and simply place the following *.envrc* file in your project folder:

```
export ANSIBLE_CONFIG=$(expand_path ansible.cfg)
```

Listing 2.2 ".envrc": "direnv" Configuration for Ansible

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

This ensures that Ansible will always access your newly created *ansible.cfg* configuration file whenever you're in the corresponding project folder or its subdirectories.

The following configuration parameters are also of interest at this point:

- `log_path`
  If desired, you can instruct Ansible to log all actions with the `log_path` directive. The log file is freely selectable, and only the directory that will contain the log file must exist. So if, for example, the *~/logs/* directory exists, you could make the following setting:

```
# [defaults]
log_path = ~/logs/ansible.log
```

  Listing 2.3 Example Excerpt from "ansible.cfg"

- `private_key_file`
  If the filename of your private SSH key deviates from the default, then you can specify it in the configuration to avoid constantly using `--private-key` on the command line:

```
# [defaults]
private_key_file = ~/.ssh/ansible_key
```

  Listing 2.4 Example Excerpt from "ansible.cfg"

2.3 Creating and Managing a Static Inventory

To inform Ansible about which hosts it should manage in the future, you can simply use an *inventory file* (also known as a *static inventory*). Your first goal should be to enable the popular and important ping test by using such a file, as follows:

```
$ ansible all -m ping
```

Feel free to try it out now; it obviously won't work because Ansible has no idea which hosts are meant by `all`.

In our *ansible.cfg*, we have already set the path to the inventory file (*./inventory*). Other filenames often seen at this point are *hosts* and *hosts.ini*. Personally, I don't like the `.ini` extension here because the file content simply does not correspond to the INI format. Also, because you might eventually have a folder named *host_vars*, it would become annoying when you pressed Tab if the inventory filename started with *ho*. (You see, when the software doesn't impose strict guidelines, personal preference decides, and that can be a matter of lively debate.)

The following is example content for the inventory file that's been tailored to our test environment:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
[test_hosts]
debian
rocky
suse
ubuntu

[test_hosts:vars]
ansible_user=vagrant
```

Listing 2.5 "inventory": First Version of Inventory File

The syntax used is not entirely definable; let's just say it's an INI-like format. But aside from that, the content should be quite understandable:

- You can define a group by putting its name (which is `test_hosts` in our example) in square brackets. You don't have to organize hosts into groups, but it's common practice and very useful. A special group called `all`, which contains all hosts, is always automatically present, and you don't need to define it.
- You can parameterize individual hosts with additional "key=value" assignments. We do not use this in this example.
- You can also parameterize host groups with the `:vars` addition (which is `test_hosts:vars` in our example). As an example application, for security reasons, we defined the account with `ansible_user`, with which Ansible should log in. (By default, the local call account is assumed, which is common with SSH.)
- Incidentally, if you're working with the atypical UNIX password authentication in your environment, you'll need to equip the corresponding inventory hosts with the additional `ansible_password=<LOGIN_PASSWORD>` parameter and also install the *sshpass* package.

Now, you perform the ping test as follows. It should be completely green:

```
$ ansible all -m ping
debian | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
ubuntu | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
}
rocky | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
suse | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.6"
    },
    "changed": false,
    "ping": "pong"
}
```

You cannot influence the order of server outputs here, and it's somewhat random due to parallel processing. More on that later.

2.4 Configuration Settings versus Parameters versus ...?

You have already learned two ways to influence Ansible's behavior: with settings in the *ansible.cfg* file and with parameters in the inventory. Often, you can achieve the same goal with different methods; for example, instead of setting `ansible_user=vagrant` in the inventory, you could have set `remote_user = vagrant` in the configuration file. (Yes—it's very unfortunate that the naming differs here!)

But that's not all by a long shot. As a third option, you could set the `ANSIBLE_REMOTE_USER` environment variable, and as a fourth option, you could use the `-u <USERNAME>` command line switch when calling `ansible`.

The reason for this variety is simply to offer users maximum flexibility. In our simple case, there was no reason to prefer one option over the other. I just wanted to show you the possibility of parameterization via inventory, which is an important technique in practice.

Of course, the different options are prioritized differently. The *https://docs.ansible.com/ansible/latest/reference_appendices/general_precedence.html* page breaks it down in detail, but I want to present it here in a simplified way. In the following list, you can see the different options in order, from *weak to strong*—meaning what's further down wins:

1. Configuration settings (*ansible.cfg*)
2. Environment variables (which, technically, also count as configuration settings)

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

3. Command line switches
4. Keywords in playbooks
5. Variables or parameters

For the moment, you can ignore option 4, and it should be noted that option 5 holds a whole sea of possibilities. (We will discuss this in Chapter 6, Section 6.1.)

If you are particularly interested in configuration settings, you should take a look at the `ansible-config` program, which offers some interesting possibilities in this regard. When invoking, you need to specify one of four subcommands: `list`, `dump`, `view`, or `init`. You can also view a separate help page for each subcommand with `--help`. The general meanings of the subcommands are as follows:

- `ansible-config list`
  The `list` subcommand shows all available configuration options, each of which includes a brief description, default setting, directive name in the *ansible.cfg*, and directive name as an environment variable.

  The overview is entirely in YAML format, which gives it a very "strict" appearance. However, it is easy to search through the overview and quickly find the desired information.
- `ansible-config dump`
  The `dump` subcommand shows all current configuration settings. Settings that differ from the default are highlighted in color, and their origin is indicated.

  With the `--only-changed` option, you can display *only* the changed settings, which is, of course, particularly interesting.
- `ansible-config view`
  The `view` subcommand simply shows the configuration file used by Ansible on the screen. The only advantage using of this subcommand instead of a `cat` command is that you don't need to know the file path for this command (as we already notedSection 2.2, there are various possibilities here).
- `ansible-config init`
  With the `init` subcommand, you can also generate templates for a configuration file (this has been available since Ansible version 2.11). Personally, I find the produced templates to be quite overloaded, so I don't use them.

2.5 Inventory Aliases and Namespaces

So far, we've assumed that the hosts in the inventory are accessible by the names listed there. For example, we've assumed that a `server57.example.org` in the inventory would also be reachable under this name for `ssh`, `ping`, and many other tools.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

But that's not always true. Perhaps your target hosts do not have names assigned, or perhaps they do have names assigned but you simply do not like them. You can compensate for all of that by using *aliases* (i.e., alternative names) in the inventory. For example, aliases for IP addresses look like this:

```
s1  ansible_host=192.168.150.10
s2  ansible_host=192.168.150.20
s3  ansible_host=192.168.150.30
s4  ansible_host=192.168.150.40
```

Listing 2.6 "inventory": Aliases for IP Addresses

And aliases for names look like this:

```
s1  ansible_host=debian.example.org
s2  ansible_host=rocky.example.org
s3  ansible_host=suse.example.org
s4  ansible_host=ubuntu.example.org
```

Listing 2.7 "inventory": Aliases for Names

Of course, again, the `ansible_host` names must be resolvable. In case your hosts are not reachable by the default port 22, the `ansible_port` parameter is available to you, as you might expect:

```
s1  ansible_host=192.168.150.10  ansible_port=2201
s2  ansible_host=192.168.150.20  ansible_port=2202
s3  ansible_host=192.168.150.30  ansible_port=2203
s4  ansible_host=192.168.150.40  ansible_port=2204
```

Listing 2.8 "inventory": Aliases with Additional Port Specifications

There are more such `ansible_*` parameters, but for the moment, these should be all you need. If you are interested, feel free to explore the **Connecting to hosts: behavioral inventory parameters** section of *https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html*.

Namespaces

If the names of your target systems are assigned according to an enumeration scheme, you don't need to list each system individually. You can work much more easily with the following range specification:

```
[webservers]
www[01:50].example.org
```

Listing 2.9 "inventory": Ranges of Hosts

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

For the single-digit numbers, you get names with leading zeros, such as `www07.example.org`. However, if you omit the leading zero in the range specification, you will not get leading zeros in the result. Ranges of letters (e.g., `[a:h]`) also work.

2.6 Beyond Ping

Making a successful ping test is an important step toward completing basic setup, but since it does not require any admin rights on the target systems, it unfortunately *cannot* show you whether the privilege escalation required for all more serious tasks is working.

And of course, it is not doing that at the moment. To prove that, you should do the following to check whether you can read the */etc/shadow* file on the target systems—something for which you undoubtedly need root privileges:

```
$ ansible all -a "test -r /etc/shadow"
debian | FAILED | rc=1 >>
non-zero return code
rocky | FAILED | rc=1 >>
non-zero return code
suse | FAILED | rc=1 >>
non-zero return code
ubuntu | FAILED | rc=1 >>
non-zero return code
```

For this to become successful, Ansible must perform a privilege escalation after authentication. In our lab environment, user `vagrant` has unrestricted, password-free `sudo` privileges, so we can achieve our goal by adding one more parameter (`ansible_become=yes`):

```
[test_hosts]
debian
rocky
suse
ubuntu

[test_hosts:vars]
ansible_user=vagrant
ansible_become=yes
```

Listing 2.10 "inventory": Second Version of Inventory File

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Just to be clear, this is already sufficient in our lab environment because user `vagrant` is equipped with password-free `sudo` rights. As a result, you'll learn about the options that are available to you if privilege escalation is organized differently in your environment.

In general, the following parameters are available to you for configuring privilege escalation:

- `ansible_become`
  This is the indication to Ansible that privilege escalation must occur on the target host. The default is `no`.
- `ansible_become_method`
  This is the method to be used. The default is `sudo`.
- `ansible_become_user`
  This is the target account for identity change. The default is `root`, which usually fits in such a context. That's why this parameter doesn't appear at all in the previous example.
- `ansible_become_pass`
  This is the password that may be required for privilege escalation.

Of course, there's something to note about the password: in a test environment, it might be acceptable to store plaintext passwords in a configuration file, but probably not in a production environment. What alternatives are available here?

- Instead, you could use the `--ask-become-pass` / `-K` option to be prompted for a password each time you run an Ansible command.
- At least in the case of `sudo`, you could configure the target hosts to skip the password prompt (e.g., for all members of the `sudo` group) as follows:

```
# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) NOPASSWD: ALL
```

Listing 2.11 "/etc/sudoers": Waiver of password authentication

Or you could do the same thing only for a specific account, as it is preconfigured on the Vagrant machines:

```
vagrant ALL=(ALL) NOPASSWD: ALL
```

Listing 2.12 "/etc/sudoers": Waiver of password authentication

Both options may not be entirely convincing, and you still wouldn't have an automatic solution for `su`. I ask you to be patient at this point, because once you know more about Ansible, you'll be able to parameterize hosts or groups in other places and even encrypt those places if necessary. You can find information about this in Chapter 10, Section 10.1.

In our test lab, we are well prepared with the current inventory for all further (administrative) actions:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
$ ansible all -a "test -r /etc/shadow"
debian | CHANGED | rc=0 >>

rocky | CHANGED | rc=0 >>

suse | CHANGED | rc=0 >>

ubuntu | CHANGED | rc=0 >>
```

2.7 A Somewhat More Complex Example

Our lab environment is admittedly quite polished: despite the fact that we have four different operating systems, we have completely uniform authentication and privilege escalation. Therefore, I would like to show you an inventory for a somewhat rougher hypothetical environment that has the following components:

- A Debian 10 system that allows direct root access with an SSH key.
- A CentOS-7 system that allows access via the `ansible` account with a password of `UserPassw0rd`. Privilege escalation is done using `su`, and we only have Python 2 available.
- An Ubuntu-18.04 system that allows access for `user1` with the SSH key from the *keys/ubuntu* file. The user has unrestricted `sudo` rights, and their password is `vmware`.

You could cover these requirements with the following inventory file:

```
[debian_hosts]
debian10   ansible_host=192.168.150.10  ansible_user=root

[centos_hosts]
centos7    ansible_host=192.168.150.20

[ubuntu_hosts]
ubuntu1804 ansible_host=192.168.150.40

[centos_hosts:vars]
ansible_user=ansible
ansible_password=UserPassw0rd
ansible_become=yes
ansible_become_method=su
ansible_become_pass=RootPassw0rd
ansible_python_interpreter=/usr/bin/python2
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
[ubuntu_hosts:vars]
ansible_user=user1
ansible_ssh_private_key_file=keys/ubuntu
ansible_become=yes
ansible_become_pass=vmware
```

Listing 2.13 "inventory": More Complex Example of Inventory

The main insights to be gained here are:

- It's usually to your advantage to use groups for parameterization.
- On the other hand, you should largely avoid parameterizing individual hosts. IP addresses are, of course, an individual matter, and for the `ansible_user` setting on the Debian system, it's not worthwhile to use a group at the moment. If there were two or more such systems, it would be different.
- More specific settings override more general settings (as seen here with `ansible_python_interpreter`), and host-specific parameters naturally have the strongest effect.

2.8 Alternative or Multiple Inventories

When the number of systems managed with Ansible grows, you will find that it is often no longer quite appropriate to manage all systems in a single inventory file. A typical example is different deployment environments such as development, testing, and production, where you always want to address all machines in one environment with the `all` Ansible identifier. Later, you'll also get to know *dynamic inventories*, which you will inevitably have to handle separately from static inventories.

In the initial stage, you could simply provide different inventory files. Of course, there can only be a maximum of one default inventory (which you have set in the *ansible.cfg*), but with any suitable Ansible command, you can also specify an inventory with the `-i` switch. Here's an example:

```
$ ansible all -i production-inv -m ping
```

Note

Do you remember the very first test calls when you had no configuration at all? Here's an example:

```
$ ansible all -i debian, -m ping
```

The `-i` switch also allows you to specify the list of target hosts directly on the command line, separated by commas. However, if you specified only a single host, Ansible would consider the hostname as a filename without the comma at the end, and the call would naturally fail.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Another way to select an inventory is by setting the `ANSIBLE_INVENTORY` environment variable, as follows:

```
$ export ANSIBLE_INVENTORY=production-inv
$ ansible […]
```

In Chapter 6, you will learn about `host_vars` and `group_vars` parameterization, which is a way to parameterize hosts through certain directories that are parallel to a given inventory. In that case, you'd want to organize different inventories in a directory hierarchy—like this:

```
ansible/
|-- inventories/
|   |-- devel/
|   |   |-- group_vars/
|   |   |-- host_vars/
|   |   `-- inventory
|   |-- testing/
|   |   |-- group_vars/
|   |   |-- host_vars/
|   |   `-- inventory
|   `-- production/
|       |-- group_vars/
|       |-- host_vars/
|       `-- inventory
|-- [...]
```

You can also use multiple inventories simultaneously. To do this, you simply apply the `-i` option repeatedly, like this:

```
$ ansible -i <inventory_1> -i <inventory_2> […]
```

When working with the environment variable, you must specify the inventories as comma-separated, like this:

```
$ export ANSIBLE_INVENTORY=<inventory_1>,<inventory_2>[,…]
```

Of course, you must avoid spaces here (or you must quote the right side accordingly).

You can even designate a parent directory as an inventory that contains any inventories at any hierarchical depth. Then, Ansible will automatically form the "sum" over all contained inventories.

Warning

Make sure that you keep track of things in such a mixture!

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Sometimes, you may have multiple equivalent inventories, which is why you don't want to specify any of them in the configuration. Instead, you should always work with the `-i` option or the `ANSIBLE_INVENTORY` environment variable.

However, you shouldn't simply not specify `inventory` in the *ansible.cfg*, but rather, you should refer to an explicitly nonexistent inventory, like this:

```
[defaults]
inventory = /dev/null
```

Listing 2.14 "ansible.cfg": Example Excerpt

The reason is that Ansible uses the */etc/ansible/hosts* inventory by default, and you might not be able to completely rule out that someone has actually entered something there—which could mean that you'd get a big surprise when your Ansible actions were unexpectedly applied to some other systems!

Now, I'd like to set aside the topic of inventories because we have other things ahead of us and we don't want to get too lost in details here. However, I'll come back to it in Chapter 12 because there is still quite a bit to say about it. For example, you will learn there that you can create static inventories in YAML format and if there are advantages to doing that. Furthermore, we'll take a look at the previously mentioned dynamic inventories.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 3
Ad Hoc Commands and Patterns

The spontaneous, interactive use of an Ansible function via the `ansible` command is referred to as an *ad hoc command* or an *ad-hoc invocation*. In later practice, this will probably become a minor matter, as you will almost always work with playbooks and the corresponding `ansible-playbook` command. Nevertheless, using ad hoc commands is a useful option that you should definitely be aware of. You can also use *patterns* to address hosts, and they are generally important and useful.

3.1 Ad Hoc Commands

You already know at least one ad hoc invocation, the ping test:

```
$ ansible all -m ping
```

Here, `ping` is one of the countless Ansible modules, `-m` is simply the short form of `--module-name`, and the `all` identifier is the pattern that specifies the set of hosts to which the ad hoc command applies. An *ad hoc command* is essentially nothing more than the invocation of exactly one module, possibly with parameterization. The general form of an ad-hoc command is as follows:

```
ansible <PATTERN> -m <MODULE_NAME> -a "[<MODULE_ARGS>]"
```

3.1.1 The "command" Module

The `command` module is certainly one of the most useful at this point, and its parameter is a Linux command. For example, you can enter the following to display the usage of the root partition on all hosts:

```
$ ansible all -m command -a "df -h /"
debian | CHANGED | rc=0 >>
Filesystem                       Size  Used Avail Use% Mounted on
/dev/mapper/debian--12--vg-root   62G  1.2G   57G   2% /
rocky | CHANGED | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda2        62G  1.5G   61G   3% /
suse | CHANGED | rc=0 >>
Filesystem      Size  Used Avail Use% Mounted on
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
/dev/sda3        64G  1.7G   58G   3% /
ubuntu | CHANGED | rc=0 >>
Filesystem                         Size  Used Avail Use% Mounted on
/dev/mapper/ubuntu--vg-ubuntu--lv   31G  3.3G   26G  12% /
```

The `command` module is actually the default, and the following call would therefore produce the same result:

```
$ ansible all -a "df -h /"
```

If you want a single-line result per host, then the `-o` / `--one-line` option can facilitate further processing or simply improve clarity:

```
$ ansible all -a "hostname" -o
ubuntu | CHANGED | rc=0 | (stdout) ubuntu
debian | CHANGED | rc=0 | (stdout) debian
rocky | CHANGED | rc=0 | (stdout) rocky
suse | CHANGED | rc=0 | (stdout) suse
```

3.1.2 The "shell" Module

If you need shell mechanisms like `|` or `>` in your call, you must use the `shell` module as follows because `command` executes the given command directly, without going through a shell:

```
$ ansible all -m shell -o -a "ps -ef | wc -l"
ubuntu | CHANGED | rc=0 | (stdout) 105
debian | CHANGED | rc=0 | (stdout) 91
rocky | CHANGED | rc=0 | (stdout) 105
suse | CHANGED | rc=0 | (stdout) 104
```

The `shell` module sends the command by default to `/bin/sh`. Although this is ultimately a `bash` on many systems, it may behave differently when called as `sh`.

If you need such a specific shell, then you have to specify it using the additional `executable` parameter, as follows:

```
$ ansible all -m shell -o -a 'echo $RANDOM executable=/bin/bash'
debian | CHANGED | rc=0 | (stdout) 21874
ubuntu | CHANGED | rc=0 | (stdout) 20953
rocky | CHANGED | rc=0 | (stdout) 26840
suse | CHANGED | rc=0 | (stdout) 1070
```

`RANDOM` is a variable that a `sh` does not normally provide. Also, you need to pay attention to quoting with single quotes—using double quotes would cause the shell on the control host to replace the variable, and the same result would come back four times.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

3.2 Use Cases Beyond "command" and "shell"

Here are some practical examples of applying ad hoc commands. You will get to know all the modules used in more detail later.

- **Rebooting hosts**

  ```
  $ ansible all -m reboot
  ```

  This module was recently added in Ansible 2.7. If you're still using an older version, you'll have to fall back to a simple `ansible all -a /sbin/reboot`. However, the behavior is different, as all connections simply break off immediately and you do not know when the hosts are back online.

- **Creating a directory**

  ```
  $ ansible all -m file -a "dest=/opt/test1 state=directory"
  ```

- **Creating a user**

  ```
  $ ansible all -m user -a "name=john shell=/bin/bash"
  ```

- **Installing a package**

  ```
  $ ansible all -m package -a "name=tree"
  ```

  (Of course, this assumes that the package has the same name on all systems.)

- **Uploading a file**

  ```
  $ ansible all -m copy -a "src=/etc/passwd dest=/tmp/test.txt"
  ```

How Am I Supposed To Know All This?

(In case you were wondering.) These are the same modules that you'll soon use daily, and eventually, you'll know the most important parameters by heart. In Chapter 7, I also provide a brief overview of the most common modules, but in practice, you'll probably work with the official documentation, for the most part. The quickest way to get there is (as usual) by using Google or another search engine of your choice. For example, try searching for "ansible user module" or "ansible copy module." The first result you get is usually the right one—at least as long as it points to something in *https://docs.ansible.com/*.

3.3 Idempotency

One characteristic that is highly valued in Ansible is *idempotency*. This term originally comes from mathematics, where it refers to the following property of functions:

$f * f = f$

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Put simply, idempotency means that it doesn't matter whether you apply a function once or twice—repeated application of the function doesn't change the result. Examples in mathematics include projections and absolute value functions.

For a configuration management system, this means that *if a configuration state has already been achieved, then all further executions of the corresponding instruction have no effect.* (In particular, there are no undesirable effects such as error messages.)

For example, let's say you need to install a software package. The first call does this, and further identical calls only determine that there is nothing to be done. This sounds trivial, but this principle applies to almost all Ansible functions, as follows:

- A configuration file already contains a desired line, and there is nothing more to do.
- A symbolic link is already present as desired, and there is nothing more to do.
- A user is already in a group, and there is nothing more to do.

You could never use a self-scripted solution to implement such smooth functionality with this level of consistency, so idempotency is one of the many conveniences of Ansible!

But enough theory—let's test the whole thing with the following call from the last section:

```
$ ansible debian -m user -a "name=john shell=/bin/bash"
debian | CHANGED => {
    "changed": true,
    "comment": "“:
    "create_home": true,
    "group": 1001,
    "home": "/home/john”:
    "name": "john”:
    "shell": "/bin/bash”:
    "state": "present”:
    "system": false,
    "uid": 1001
}

Then, we do exactly the thing same again:
$ ansible debian -m user -a "name=john shell=/bin/bash"
debian | SUCCESS => {
    "append": false,
    "changed": false,
    "comment": "“:
    "group": 1001,
    "home": "/home/john”:
    "move_home": false,
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    "name": "john”:
    "shell": "/bin/bash”:
    "state": "present”:
    "uid": 1001
}
```

In our current discussion, the value of `changed` is the crucial value here. On the first call, there was a change, but there wasn't in the second. (You need to get used to this jargon!)

3.4 Internal Workings

To understand some terms, you need to look behind the scenes of Ansible—but you don't need to do it just yet. You can treat Ansible as a black box and still work with it successfully.

3.4.1 Parallel Execution

Ansible generally accesses its hosts in parallel for each task. However, the number of *forks* (i.e., simultaneous connections) is set conservatively by default to five. You can also dynamically specify the number of forks you want with the `-f` option on the command line:

```
$ ansible all -f 10 -m ping
```

Alternatively, you can persist it in the configuration in the `[defaults]` section:

```
# [defaults]
forks = 10
```

Listing 3.1 "ansible.cfg": Excerpt for Increasing the Number of Forks

In practice, 50 or more forks are quite common, and the key is that your control host and your network must be able to handle it.

In our test lab, however, you will hardly notice any difference by changing this value. At most, you could cause a slowdown in execution by setting the number of forks to 1 (just for fun).

3.4.2 Persistent Connections

You already know that Ansible addresses its hosts via SSH by default. To achieve adequate performance with this (because the establishment and teardown of SSH connections does matter), Ansible works with *persistent connections*.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Let's make these visible. You can do this by using tools like `ss` or `lsof`, at least one of which should be available. You should also perform the following Ansible action first on the target hosts because the connections are typically cleared after sixty seconds of inactivity:

```
$ ansible all -m ping
[...]

$ lsof -i :22
COMMAND  PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
ssh     6884 vagrant    3u  IPv4  61544      0t0  TCP
        192.168.150.100:33674->rocky.example.org:ssh (ESTABLISHED)
ssh     6887 vagrant    3u  IPv4  62611      0t0  TCP
        192.168.150.100:54188->ubuntu.example.org:ssh (ESTABLISHED)
ssh     6891 vagrant    3u  IPv4  61552      0t0  TCP
        192.168.150.100:35210->suse.example.org:ssh (ESTABLISHED)
ssh     6896 vagrant    3u  IPv4  61533      0t0  TCP
        192.168.150.100:45816->debian.example.org:ssh (ESTABLISHED)

$ ss -tn
ESTAB   0   0   192.168.150.100:45816   192.168.150.10:22
ESTAB   0   0   192.168.150.100:33674   192.168.150.20:22
ESTAB   0   0   192.168.150.100:35210   192.168.150.30:22
ESTAB   0   0   192.168.150.100:54188   192.168.150.40:22

$ ls -l ~/.ansible/cp/
srw------- 1 vagrant vagrant 0 Oct 31 12:00 4da0326269
srw------- 1 vagrant vagrant 0 Oct 31 12:00 8082b77a5c
srw------- 1 vagrant vagrant 0 Oct 31 12:00 8c04ad47d2
srw------- 1 vagrant vagrant 0 Oct 31 12:00 c465e60c60
```

(Note that in this example, we have slightly modified the command outputs due to space constraints.)

The final directory listing shows the socket files that SSH uses (`cp` stands for *control path*).

In very rare cases, persistent connections can cause problems. This often happens when changes are made to the firewall settings of the target system using Ansible, causing connections to die. This makes Ansible simply receive any response from the corresponding target host, even though everything is already back to normal from a network perspective (including normal pings and SSH).

When this occurs, the remedy is to either to wait a short while or to locate and terminate the hanging SSH processes, as follows:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
$ lsof -i :22
COMMAND   PID   USER   FD   TYPE  DEVICE SIZE/OFF NODE NAME
[...]
ssh     12314  user1   3u   IPv4 1249038      0t0 TCP [...]-><host>:ssh
[...]

$ kill -HUP 12314
```

However, since unused connections are usually cleaned up fairly quickly by default, simply waiting is probably more efficient. By the way, you can also do this in "high-level" fashion (provided that the corresponding inventory is available) with the following ad hoc call:

```
$ ansible all -m meta -a reset_connection
```

The topic of connections is much more extensive than we can discuss at this point. Everything is designed modularly, which allows Ansible to handle many other connection techniques like Docker and Windows Remote Management (WinRM). For more information on this topic, please take a look at *https://docs.ansible.com/ansible/latest/plugins/connection.html*.

3.4.3 What Happens When a Module Is Called?

So far, we know that Ansible (typically) works over SSH and requires nothing more on the target hosts than a Python interpreter (in principle). Therefore, all Ansible modules exist only on the control host but must be executed on a target host—but how does that work?

To explore this, you start a `ping` call again, this time with verbosity level of 3 (`-vvv`). This generates a lot of informative additional output, but we will only extract some relevant lines in the following listing (the output has been significantly shortened and adjusted):

```
$ ansible debian -m ping -vvv
[...]
Using module file /usr/lib/python3/dist-packages/ansible/
    modules/system/ping.py
<debian> SSH: EXEC ssh [...] mkdir -p /home/ansible/.ansible/tmp/
    ansible-tmp-1587806016.888666-214053797275925 [...]
<debian> PUT /home/ansible/.ansible/tmp/
    ansible-local-53838zlrvng8/tmpu3gowgkr
         TO /home/ansible/.ansible/tmp/
    ansible-tmp-1587806016.888666-214053797275925/AnsiballZ_ping.py
[...]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
<debian> SSH: EXEC ssh [...] /home/ansible/.ansible/tmp/
    ansible-tmp-1587806016.888666-214053797275925/AnsiballZ_ping.py [...]
[...]
<debian> SSH: EXEC ssh [...] rm -f -r /home/ansible/.ansible/tmp/
    ansible-tmp-1587806016.888666-214053797275925/ [...]
[...]
```

From this, you can infer what is happening behind the scenes:

1. Ansible creates a temporary working directory on the target system.
2. Ansible generates a standalone executable from the called module (which is `ping` in this example) that contains all the necessary information and parameters for execution.
3. This executable is transferred to the target system into the temporary working area, and it starts there.
4. Finally, the temporary working area is removed again.

Now, you should have a rough idea of how Ansible works. If you ever want to know how it works in *much* more detail, you can always look into Ansible's source code. But even if you want to develop your own modules at some point, you don't need overly detailed knowledge of Ansible's internal processes.

Now, let's refocus on more practical things!

3.5 The Ansible Console

If you want to work interactively with various target hosts for an extended period, you can use the `ansible-console` command, which provides an integrated shell environment:

```
$ ansible-console
Or, if there are color issues, you can use this command:
$ ANSIBLE_NOCOLOR=1 ansible-console
```

The first word on each command line is interpreted as an Ansible module. If no such module exists, then the entire line is executed instead with the shell module as a Linux command:

```
vagrant@all (4)[f:5]$ ping
[...]

vagrant@all (4)[f:5]$ df -h /
[...]
```

- You then exit the console with `exit` or by pressing Ctrl+D, as is usual in UNIX.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

- In the rare cases when you want to execute a command that has the same name as a module (e.g., `hostname`), it's easiest to prefix it with `command` or `shell`.
- With the built-in `cd` command, you can "switch" into individual target hosts or groups. A `cd` without an argument expands the addressing back to all hosts.
- With the built-in `list` command, you can see the hosts you are currently addressing.

Color Schemes

If you (like me) prefer working with white terminal windows, then the default white text of the console is somewhat inconvenient. Since version 2.7, you've been able to set the text color of the prompt in the configuration file, in the `[colors]` section. So, you can add something like this if needed:

```
# ...

[colors]
console_prompt = black
```

Listing 3.2 "ansible.cfg" Excerpt for Adjusting Text Color of Ansible Console

3.6 Patterns for Addressing Hosts

Until now, we have generally referred to all hosts in our inventory with the `all` identifier. However, with Ansible, you can also direct your actions toward specific hosts or groups, as follows:

```
$ ansible suse       -m ping
$ ansible test_hosts -m ping
```

(As already mentioned, `all` is just a group that's always available by default, and you don't need to explicitly define it.)

If your inventory hosts are currently unreachable or you want to test the behavior of patterns with purely fictional entries, it is better to use these commands:

```
$ ansible suse       --list-hosts
$ ansible test_hosts --list-hosts
```

Certain special characters also provide other selection options. For example, the usual `*` and `?` wildcards are available, and you should quote them as follows because of the shell:

```
$ ansible '*'    -m ping
$ ansible 'su*'  -m ping
$ ansible 's??e' -m ping
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

You can also form unions of hosts or groups with a comma or colon:

```
$ ansible 'debian,rocky' -m ping
$ ansible 'debian:rocky' -m ping
```

If an exclamation mark precedes a pattern, then it is understood to be an exclusion. You can also apply exclusions multiple times—for example, as follows:

```
You do it this way for all except suse:
$ ansible 'all:!suse' -m ping

And you do it this way for all except suse and ubuntu:
$ ansible 'all:!suse:!ubuntu' -m ping
```

Regular expressions are also available for advanced users. Such a pattern must start with a tilde (~), as follows:

```
$ ansible '~^ub.*u$' -m ping
```

This example pattern matches all hosts and groups whose name starts with `ub` and ends with `u`. All possibilities are described at *https://docs.ansible.com/ansible/latest/user_guide/intro_patterns.html*.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 4
YAML: The Configuration and Serialization Language

This chapter is about the YAML language—from a pragmatic perspective. You need to learn this language first if you want to work productively with Ansible (or Docker Compose, or Kubernetes or ...). Syntactically, YAML is a quite simple language for data serialization, and it's mostly used as a "configuration language." The acronym *YAML* stands for *YAML Ain't Markup Language* (though it originally stood for *Yet Another Markup Language*).

4.1 Syntax and Structure

YAML exhibits the following characteristics regarding syntax and structure:

- By convention, YAML files have the `.yml` or `.yaml` file extension.
- YAML files are encoded in UTF-8 or UTF-16.
- Comments are possible with the usual # symbol.
- The structure of a YAML document is primarily defined by *indentation*.
- Single values (scalars), lists (arrays), and maps (associative arrays or dictionaries) and any nested combinations thereof can be represented in YAML.
- If possible, scalar text values can be noted simply as they are; when you need to, you can quote them with `"..."` or `'...'`.

You can find a basic example of how YAML "feels" on the YAML project's website (see *https://yaml.org*), where the page is presented in syntactically correct YAML.

The three dashes (`---`) that are often seen at the beginning of a YAML file signal the start of a new document, and they also allow for specifying certain meta-information, such as the YAML version used. The latter is usually unnecessary in practice, and the three dashes are very often omitted. However, if your file is supposed to contain multiple documents, then they become mandatory as separators. They are not used at all in Ansible, but they are used a lot in Kubernetes.

Three dots (`...`) mark the end of the document, but we can safely ignore them for now.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

4.2 Editing YAML Files

For more complex YAML documents, we recommend that you use an editor with YAML support, which should offer not only syntax highlighting but also the smartest indentation assistance possible. I would like to briefly introduce some candidates to you.

You could test your editor's YAML capabilities with the following file:

```
---
animals:
  - Dog
  - Cat
data:
  int: 42
  float: 3.1415
  string: "Hello"
  bool: true
list_text: >
  {{ list | join(', ') }}
more_text: this may be incorrectly colored
```

Listing 4.1 "yaml-test.yml": YAML Test File

4.2.1 vim

The `vim` editor is the favorite tool of many admins, but in its default state, `vim` is usually not quite YAML ready. We recommend that you make the following adjustments to `vim` for general and productive work with YAML files:

- The syntax highlighting is sometimes faulty or not implemented intelligently enough. For example, you can check this with the file from Section 4.2 (focus on the last line). If this bothers you and you want to see some colors, then you can comment out the two sections in the */usr/share/vim/vim*/syntax/yaml.vim* file that begin with `syn region yamlFlowString`. (There are six lines of text in a row.) That way, you'll see fewer colors, but they'll be in the right places.

  Note that the comment character here is the double quotation mark!

- With the following *.vimrc* settings, you should be able to work well with YAML files:

```
syntax on
filetype plugin on
set pastetoggle=<F2>
autocmd FileType yaml setlocal tabstop=2 softtabstop=2 shiftwidth=2
                           \ expandtab autoindent indentexpr=
```

Listing 4.2 "~/.vimrc": Example vim Configuration for YAML Files Without Specific Plug-in

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The essential aim here is to configure a consistent indentation depth of two spaces, which allows you to easily change the levels by pressing the Tab or Backspace key.

There are also specific plug-ins for vim that support the specific "language" beyond general YAML support. For example, among Ansible users, the *ansible-vim* plug-in (*https://github.com/pearofducks/ansible-vim*) is quite well known. Experts in vim who want to try something like this will surely be able to manage with the explanations on the project page—but personally, I've found that such plug-ins are almost always "too smart" for me, which is why I don't necessarily recommend that you use them here.

To conclude working with vim, here's a tip: if you ever have trouble with any invisible tabs or spaces, switching to :set list helps because it reveals these nonprintable characters.

4.2.2 nano

Unfortunately, the popular nano editor usually doesn't offer *out-of-the-box* support for YAML. In fact, older often versions don't include syntax highlighting for YAML at all.

So, you need to first test whether your nano colorizes a YAML file. If it doesn't, you can find suitable syntax highlighting files on GitHub that you can download into your */usr/share/nano/* directory. (The global configuration of nano includes all files located there.) If you want to add some color, this download command will suffice:

```
# curl -sSL https://raw.githubusercontent.com/serialhex/nano-highlight/\
master/yaml.nanorc \
-o /usr/share/nano/yaml.nanorc
```

Now, all that's missing is some support for proper indentation of YAML code. Here are some sensible settings you can include in your personal *~/.nanorc*:

```
set autoindent
set tabsize 2
set tabstospaces
```

Listing 4.3 "~/.nanorc": Example nano Configuration for YAML Files

The autoindent setting ensures that you'll keep the indentation level when starting a new line, and the other directives are self-explanatory. One catch remains: there seems to be no way to link these settings *only with YAML files*. If this bothers you, then you could always call nano with the nano -iET2 <FILE.yml> command for YAML files or create an alias like yamlnano to make it easier.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

4.2.3 Emacs

Emacs with the YAML mode offers excellent YAML support *out of the box*. (Note that Debian/Ubuntu packages *emacs* and *yaml-mode*.) But just like `vi`, *Emacs* requires some getting used to. Even if you use the X11 variant with a graphical interface, Emacs is definitely not an editor you can switch to easily!

If your distribution does not come with a prepackaged YAML mode (like SUSE), then you can download the *.el* file from *https://github.com/yoshiki/yaml-mode* and place it in */usr/share/emacs/site-lisp/*.

To activate YAML mode automatically when loading YAML files, you can simply add the following entry to your *.emacs* file:

```
(require 'yaml-mode)
(add-to-list 'auto-mode-alist '("\\.yml\\'" . yaml-mode))
```

Listing 4.4 "~/.emacs": Activation of YAML Mode When Loading YAML Files

4.2.4 Notepad++

The very popular Windows Notepad++ editor is also very well suited to editing YAML files. If you want to access Linux files transparently with it, then you can either activate the *NppFTP* plug-in or use *WinSCP*, which also allows transparent editing.

You should adjust the handling of tabs before you start working with YAML files. To do this, you select **Settings • Preferences** from the menu and then select the **Indentation** submenu. Then, under **Indent Settings**, you select the **yaml** language and do the following:

- Set the **Indent size** to 2.
- Select the **Space character(s)** radio button.

4.3 Syntactic Checks

You can use `yamllint` to perform a general syntax check of YAML files. On Debian-based systems and Rocky Linux (EPEL), `yamllint` is part of the *yamllint* package, and on SUSE, it's found in the *python3-yamllint* package.

The invocation is quite simple:

```
$ yamllint <FILE.yml>
```

We highly recommend that you use the `yamllint` tool because it not only checks the bare syntax but also points out certain "cosmetic" details that might cause problems when you're actually using the YAML file.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

If there is no output, then everything is fine (as is common in Linux).

Another interesting tool is the `yq` YAML processor, which supports many other formats besides YAML. If you're interested, you should download it from the project page *https://github.com/mikefarah/yq* because it is packaged in very few distributions (and even when it is, it's usually an outdated version).

The focus of this tool is more on processing files, such as extracting information selectively. But of course, it also supports simple validation, as follows:

```
$ yq <FILE.yml>
```

4.4 Lists and Maps

Now, I'll provide some lists and maps as brief examples of the essential syntactic elements of YAML.

Lists—that is, ordered collections of things—are represented in YAML as follows:

```
- Dog
- Cat
- Mouse

# And here's how they're represented using different syntax:
[Dog, Cat, Mouse]
```

Listing 4.5 Representation of Lists in YAML

Maps (associative arrays) are collections of key-value pairs, and they are represented as follows:

```
Dog: Woof
Cat: Meow
Mouse: Squeak

# And here's how they're represented using different syntax:
{Dog: Woof, Cat: Meow, Mouse: Squeak}
```

Listing 4.6 Representation of Map in YAML

4.5 Nested Structures

In YAML, you can create any combination of maps and lists—but be sure to pay attention to the level of indentation! (The exact indentation *depth* is not important. Two spaces per level are common.)

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Here's a list of lists:

```
-
  - Lassie
  - Fido
-
  - Kitty
  - Puss
```

Listing 4.7 List of Lists in YAML

However, you'll rarely encounter such a combination, because in practice, lists of maps are always preferred.

Here's a list of maps:

```
- Dog: Lassie
  Sound: Woof

- Cat: Kitty
  Sound: Meow
```

Listing 4.8 List of Maps in YAML

And here's a map with lists:

```
Lassie:
  - Woof
  - Growl

Kitty:
  - Meow
  - Purr
```

Listing 4.9 Map with Lists in YAML

And finally, here's a map with maps:

```
Dog:
  Name: Lassie
  Age: 7
  Food: Pedigree

Cat:
  Name: Kitty
  Age: 9
  Food: Whiskas
```

Listing 4.10 Map with Maps in YAML

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

4.6 Text Passages and Block Expressions

If you want to specify texts with line breaks in YAML, then *block expressions* offer an interesting option.

Here's a typical example—a longer text passage as a map value:

```
text: |
  Dear Reader,

  I hope you are doing well!

  Kind regards,
    Axel Miesen
```

Listing 4.11 Block Expression in YAML

The entire text is aligned with its first line, and indentations are preserved.

The variant with > replaces all newlines between lines (except the last one) with a space:

```
text: >
  This is simply a very long sentence,
  but it should not be wrapped in the middle!
```

Listing 4.12 Block Expression without Line Breaks in YAML

To create a line break in the result of this version, you simply insert a blank line in the source text.

4.6.1 Variations

There are variations of these block expressions that allow you to control the trailing newline and subsequent blank lines. Table 4.1 provides an overview of them.

Expressions	Meanings
| or >	Keep the trailing newline and remove trailing blank lines.
|- or >-	Remove both.
|+ or >+	Keep both.

Table 4.1 Variants of YAML Block Expressions

By the way, it's somewhat tricky to indent the entire result text. For example, YAML lets you add a number after the expressions described in the table to set the indentation level:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
content: |2
     This text is indented
     by three spaces.
```

Listing 4.13 Block Expression with Indentation Control

The `2` is the information for YAML that *everything from depth two* is part of the text.

4.6.2 Long Texts without Line Breaks

If, for the purpose of clarity, you want to write a long text that extends over several lines (e.g., a very long URL), then you won't want to have spaces between the individual parts of it. Here, escaping the newlines helps:

```
text: "\
  Dog\
  Cat\
  Mouse\
  "
```

As a result, the text would look like this:

```
DogCatMouse
```

4.7 Nothingness in YAML

Just like many other languages, YAML provides the ability to specify a nonexistent value. For example, you may be familiar with the `NULL` value in SQL, the `undef` value in Perl, and the `None` value in Python. However, if you're not yet familiar with this concept, just imagine you have to measure temperatures at certain times, and in between, the sensor occasionally fails and so nothing is measured. At those times, when making a list of measurements, you would want to record that *no* value was measured.

Syntactically, in YAML, you have several options to express this:

```
value1: # nothing here
value2: ~
value3: null
value4: Null
value5: NULL
```

Listing 4.14 Nonexistent Values in YAML

Of course, the comment after `value1` is not mandatory, and it only serves to improve readability.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

4.8 Anchors and References

When entire sections of YAML code are repeated, you can insert anchors and references that point to them to avoid redundancy:

```
animals: &some_animals
  Dog: Woof
  Cat: Meow
  Mouse: Squeak

animals_again: *some_animals
```

If necessary, you can not only clone the referenced object but even extend it:

```
animals: &some_animals
  Dog: Woof
  Cat: Meow
  Mouse: Squeak

more_animals:
  <<: *some_animals
  Donkey: hee-haw
  Duck: quack
```

Depending on the parser you use, you can also overwrite existing entries with this technique. Ansible supports this, but you will see a warning or an informational message.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 5
Playbooks and Tasks: The Basics

Playbooks are the entry points for any Ansible project. In the simplest case, they contain an ordered set of individual steps (*tasks*) that describe how to achieve the desired configuration of the target hosts.

5.1 Hello Ansible: The Very First Playbook

Although Ansible is not a programming language in the strictest sense, we want to follow the good-old tradition and create a "Hello World" playbook as our first official act. Playbooks are written in YAML, and they conventionally have the `.yml` or `.yaml` file extension. The playbook in Listing 5.1 is pretty much the simplest one possible:

```
---
- hosts: localhost

  tasks:
    - debug: msg="Hello Ansible!"
```

Listing 5.1 "hello-ansible.yml": Very First Playbook

Warning

If you are not sufficiently familiar with the YAML format, you *really* should read Chapter 4 first.

The essential thing with YAML is the correct indentation of text blocks—you must not do this in a haphazard way!

To ensure that your project folder isn't cluttered over time, I suggest that you create a *playbooks/* folder for playbooks as follows:

```
$ cd ~/ansible/projects/start
$ mkdir playbooks
```

Then, you should store the playbook file in this new folder.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Before we start the whole thing, we'll explain the content:

- The three dashes are a known YAML marker that indicates the beginning of the document. Many Ansible users omit them, and that's not a problem since having multiple YAML documents in one file is not possible with Ansible anyway.

  For completeness, Ansible uses the Python PyYAML module as a parser, which currently implements YAML version 1.1.
- The `hosts` keyword specifies the target hosts to address. You can use any pattern instead of `localhost` (Chapter 3, Section 3.6), and very often, you'll see the group name `all` to address all inventory hosts.

  You can also provide a list of patterns, like this one:

  ```
  - hosts:
      - debian
      - rocky
  ```

  Or you can provide a list of patterns in the compact YAML list form:

  ```
  - hosts: [debian, rocky]
  ```

  Or you can even provide a list of patterns as one Ansible pattern string:

  ```
  - hosts: debian, rocky
  ```
- The `tasks` keyword indicates that the subsequent steps transform the target hosts into the desired state.
- The call to the `debug` module is currently our only task; the `msg` parameter provides the desired output.

At first glance, it might not be entirely clear to you why the playbook begins with a list. (Technically speaking in YAML, the content of our playbook is a list with a single element!) The explanation is that a playbook can consist of several so-called *plays*, each of which consists of a set of hosts to which certain tasks are assigned. However, you will not need playbooks with more than one play until much later.

To start the whole thing (at last), you use the `ansible-playbook` command and pass the path to the playbook as a parameter. If your current working directory is the root folder of your project, then it looks like this:

```
$ ansible-playbook playbooks/hello-ansible.yml

PLAY [localhost] ***********************************************************

TASK [Gathering Facts] *****************************************************
ok: [localhost]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
TASK [debug] *******************************************************************
ok: [localhost] => {
    "msg": "Hello Ansible!"
}

PLAY RECAP *********************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0
                             skipped=0    rescued=0    ignored=0
```

The output requires some clarification:

- The output begins with `PLAY` and the list of the corresponding target systems (in our example, this is only `localhost`). You can also provide a `name:` attribute to a play, and the value of this attribute would be displayed here. However, if the playbook contains only a single play, then you'd typically skip providing a `name:` attribute.
- The tasks follow with a brief description and a corresponding result.
- We didn't request the `Gathering Facts` task, but it is Ansible's default behavior to gather information about the involved target systems at the beginning of a play. I will discuss this in detail in Chapter 6, Section 6.3. In most real playbooks, you need these facts, but in our simple example, you could save some time by specifying `gather_facts: no` or `gather_facts: false` to skip this step, as follows:

  ```
  - hosts: localhost
    gather_facts: no
  ```

- The last line provides a summary report: in the `localhost` target system, two tasks were completed with the desired (successful) result (as indicated by `ok=2`), there were no changes (`changed=0`), the target system was reachable (`unreachable=0`), and no tasks failed (`failed=0`). For now, you can ignore the `skipped`, `rescued`, and `ignored` fields.

5.1.1 Relative Paths in Playbook Calls

In our chosen directory structure, you must live with the minor inconvenience of having to make all calls from the root folder of the project, as follows:

```
$ ansible-playbook playbooks/hello-ansible.yml
```

You must do this unless (as recommended in Chapter 2, Section 2.2) you use the *direnv* software or a similar clever method that gives you more flexibility. From now on, I will omit the *playbooks/* path component in the examples to spare you this redundancy:

```
$ ansible-playbook hello-ansible.yml
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

5.1.2 Addressing the Actual Target Hosts

The fact is that `localhost` is not a common target for administrative tasks, so let's address all the actual targets in the test lab:

```
---
- hosts: all

  tasks:
    - debug: msg="Hello Ansible!"
```

Listing 5.2 "hello-all.yml": Hello to Everyone

And to make things a bit more exciting, you can shut down a machine (e.g., `suse`) beforehand (by using `vagrant halt suse` in the *vagrant/* folder). The result will be something like this:

```
$ ansible-playbook hello-all.yml

PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [rocky]
ok: [debian]
ok: [ubuntu]
fatal: [suse]: UNREACHABLE! => {"changed": false, "msg":
"Failed to connect to the host via ssh: ssh: connect to host suse port 22:
Connection timed out": "unreachable": true}

TASK [debug] *******************************************************************
ok: [debian] => {
    "msg": "Hello Ansible!"
}
ok: [rocky] => {
    "msg": "Hello Ansible!"
}
ok: [ubuntu] => {
    "msg": "Hello Ansible!"
}

PLAY RECAP *********************************************************************
debian                     : ok=2    changed=0    unreachable=0    failed=0
rocky                      : ok=2    changed=0    unreachable=0    failed=0
suse                       : ok=0    changed=0    unreachable=1    failed=0
ubuntu                     : ok=2    changed=0    unreachable=0    failed=0
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

First, you'll notice that there seems to be no fixed order of target systems per task. Due to parallel processing, the order is indeed quite random; we already discussed this in Chapter 3, Section 3.4.1.

You should also note that `suse` was completely out of the game after its inaccessibility was determined, as the second task did not even attempt to contact this host.

Finally, the concluding statistics should not pose any puzzles here either. (I have abbreviated the output slightly to save space.) If you have followed the example in the lab, then remember to let the powered-off machine participate again (by using `vagrant up suse`).

5.2 Formulation of Tasks

Most tasks consist of calling an Ansible module, possibly with appropriate parameterization.

Note

For the sake of clarity, let me state this once more: you can only call *one* module per task, and if your work requires two (or more) module calls, then you will need two (or more) tasks.

Apart from that, Ansible does allow certain freedoms in the formulation of tasks—which is all the more reason to agree with your team on a style guide or some best practices as early as possible.

The most important thing is that each task gets a `name` attribute. With a `debug` task, you don't have to be quite so strict, but giving a `name` to most other types of tasks should be a hard and fast rule. Inserting a blank line between individual tasks also doesn't hurt.

Here is an example of how *not* to do it better:

```
tasks:
  - debug: msg="Hello"
  - command: df -h /
  - service: name=sshd state=started
```

Listing 5.3 Three Tasks in Unattractive Style

It would be much better like this (since the task names also appear in the output during the playbook run, which greatly facilitates overview and orientation):

```
tasks:
  - name: Some test output
    debug: msg="Hello"
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
- name: Determine root file system space usage
  command: df -h /

- name: Start SSH service
  service: name=sshd state=started
```

Listing 5.4 Three Much More Readable Tasks

Note

These tasks are intended purely as examples of good and not-so-good style. We will discuss what `command`- and `service`-calls actually do in a moment.

The last of these three tasks is a typical example of a *parameterized module call*, in which you can specify the parameters as `<name1=value1> <name2=value2> ...`.

To avoid overly long lines, you'll sometimes use a YAML block expression:

```
- name: Start SSH service
  service: >
    name=sshd
    state=started
```

Alternatively, you can and should specify the parameters as a YAML map. You should also pay close attention to the indentation levels:

```
- name: Start SSH service
  service:
    name: sshd
    state: started
```

Tip

Using a YAML map to define module parameters should be your method of choice—at least as soon as you want to pass two or more of them!

I also want to point out again that a map as a data structure is an *unordered* set of key-value pairs. Therefore, from a technical point of view, the order you write key-value pairs in doesn't matter at all. Here, you see three examples that are completely identical internally:

```
- name: Start SSH service
  service:
    name: sshd
    state: started
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    - name: Start SSH service
      service:
        state: started
        name: sshd

    - service:
        state: started
        name: sshd
      name: Start SSH service
```

However, the whole thing is *indeed* again a matter of taste—and in your team, it'll also very quickly become a matter of which style guide you use. Placing a `name` attribute at the bottom is certainly quite unusual, and you would quickly become unpopular if you did so.

5.3 Ending Plays

If, usually for testing purposes, you want to end a play prematurely, then you can use the following instruction:

```
    - meta: end_play
```

You can place it at the same indentation level as a normal task, and the play will end at that point.

5.4 The Problematic Colon

Sooner or later, you'll want to use a colon somewhere—for example, in a debug message or in a task name (as in the example from Listing 5.5):

```
---
- hosts: localhost

  tasks:
    - name: Tell me: What's the problem here?
      debug: msg="This won't work"
```

Listing 5.5 "error-colon.yml", Faulty Playbook

Since a colon followed by a space has a special meaning in YAML, you will encounter a syntax error here:

```
ERROR! Syntax Error while loading YAML.
[...]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
The offending line appears to be:

  tasks:
    - name: Tell me: What's the problem here?
                  ^ here
```

If you cannot or do not want to avoid colons, then simple quoting like "..." or '...' often helps. However, if the `msg` parameter of the `debug` module contains a colon, then you must double-quote it like this for internal reasons:

```
---
- hosts: localhost

  tasks:
    - name: "Tell me: What's the problem here?"
      debug: 'msg="Now it works: Yay!"'
```

Listing 5.6 "colon.yml" with All Colons Placed Correctly

Of course, you can also swap the placement of double and single quotes—but in this case, it's easier to choose the YAML map for the parameters because simple quoting will suffice:

```
---
- hosts: localhost

  tasks:
    - name: "Tell me: What's the problem here?"
      debug:
        msg: "Now it works: Yay!"
```

Listing 5.7 "colon2.yml" with Everything Correct

5.5 Error Handling and Retrying Files

Errors happen, even if you do everything right. In this section, we want to explain how Ansible handles errors.

There are two types of errors:

1. Parse-time errors
2. Runtime errors

Parse-time errors are errors that Ansible can detect before the playbook even starts running properly. They include typos in module names or other keywords, incorrect YAML indentation (which is very common), and similar issues. This type of error is actually

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

quite boring: in general, Ansible provides helpful explanations, you fix the error, and then, you try again.

On the other hand, the truly relevant errors are *runtime errors*. They occur when, on some host, a task cannot be successfully executed due to a severe error or exception. By default, Ansible behaves as follows in the event of a runtime error:

Behavior In Case of Runtime Errors

If a task fails on a host, that host is excluded from all further actions. The play continues for the remaining hosts that have been error-free up to this point.

You can test this with the following small playbook from Listing 5.8, which will only be successful on Debian-like systems. (If you cannot distinguish between Debian and non-Debian systems in your inventory, then please use any other command that can only be successful on some of your target hosts.)

```
---
- hosts: all
  gather_facts: no

  tasks:
    - name: Search for "debian" in /etc/os-release
      command: grep -i debian /etc/os-release

    - debug: msg="Let's continue..."
```

Listing 5.8 "fail-on-non-debian.yml": Which Fails on Non-Debian Systems

Here is the effect (with shortened output):

```
$ ansible-playbook fail-on-non-debian.yml

PLAY [all] *********************************************************************

TASK [Search for "debian" in /etc/os-release] **********************************
changed: [ubuntu]
changed: [debian]
fatal: [rocky]: FAILED! => {"changed": true, "cmd": ["grep", "-i", "debian",
fatal: [suse]: FAILED! => {"changed": true, "cmd": ["grep": "-i", "debian",

TASK [debug] *******************************************************************
ok: [debian] => {
    "msg": "Let's continue..."
}
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
ok: [ubuntu] => {
    "msg": "Let's continue..."
}

PLAY RECAP *********************************************************************
debian                     : ok=2    changed=1    unreachable=0    failed=0
rocky                      : ok=0    changed=0    unreachable=0    failed=1
suse                       : ok=0    changed=0    unreachable=0    failed=1
ubuntu                     : ok=2    changed=1    unreachable=0    failed=0
```

If an error occurs, Ansible can create a file named *<PLAYBOOKNAME>.retry* that contains the names of all failed hosts. You can then use these to limit a subsequent playbook run if needed (by using option `--limit @<RETRYFILE>`), which significantly reduces the number of target hosts. After all, there's no need to bother the hosts where everything has already run successfully again, and even determining idempotency puts a slight load on systems.

There are two configuration directives that control this behavior:

- The `retry_files_enabled` configuration directive determines whether retry files are generated at all (default from Ansible 2.8: `false`, previously `true`).
- The `retry_files_save_path` configuration directive specifies a directory where the retry files are written (which, by default, is next to the playbook).

My recommendation is that retry files are generally not bad, but the default storage location is sometimes inconvenient. If you want to enable this, you could configure Ansible as follows:

```
# [defaults]
retry_files_enabled = yes
retry_files_save_path = ~/.ansible/retry-files
```

Listing 5.9 "ansible.cfg" Excerpt for Enabling Retry Files

5.6 Tags

Sometimes, you'll want to run only parts of a playbook instead of the whole playbook. For example, if you're currently developing the thirty-seventh task of a playbook, you'll find it quite tedious to always have to wait for the first thirty-six tasks to complete during a test run.

One thing you can do in Ansible is to optionally mark tasks with *tags*, as shown in the following example in Listing 5.10:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- hosts: localhost
  gather_facts: no

  tasks:
    - name: Say hello
      debug: msg="Hello Ansible!"
      tags: hello

    - name: Another message
      debug: msg="Everything is fine so far"
      tags: [test, message]

    - name: That's it
      debug: msg="Finished"
      tags: always
```

Listing 5.10 "tags.yml" Example Using Tags

In the second task, you can see that using lists of tags is also possible. The special `always` tag ensures that the corresponding task is always executed (unless it is explicitly skipped). Now, possible calls could be as follows:

```
All tasks marked with "hello":
$ ansible-playbook tags.yml -t hello

All tasks marked with "hello" or "test":
$ ansible-playbook tags.yml -t hello,test

All tasks except those marked with "message":
$ ansible-playbook tags.yml --skip-tags message
```

Later, this can become a bit tricky once you develop tasks that build on each other. For instance, if an earlier task generates a variable that is used by a later task, then it's only logical that you can't use the later task independently.

Note

When you're working with tags, you should always make sure to tag *all* tasks you need appropriately!

You have already seen the special `always` tag, and Table 5.1 shows all the special tags that are available:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Tags and Invocations	Meanings
Tag: `always`	The task is always executed (unless it is explicitly skipped).
Tag: `never`	The task is never executed (unless it is explicitly requested).
Invocation: `ansible-playbook [...] -t tagged`	Execute only tasks that have at least one tag.
Invocation: `ansible-playbook [...] -t untagged`	Execute only tasks that have no tag.

Table 5.1 Special Tags and Invocations

5.7 The "ansible-playbook" Command

The `ansible-playbook` command is the main tool you'll use when working with Ansible (at least as long as you use the command line). Like any Linux command, it offers a variety of options, and I will now provide an overview of the most important ones.

A general invocation of `ansible-playbook` reads as follows:

```
ansible-playbook [OPTIONS] playbook.yml [playbook2.yml ...]
```

You can also invoke `ansible-playbook` with multiple playbooks, although this is rather rare in practice. All included plays are executed in the corresponding order, and later plays can also access the results of earlier plays.

Table 5.2 provides an overview of the most useful options.

Options	Meanings
`-l` <HOST_OR_GROUP>	Restrict actions to certain hosts.
`--list-hosts`	This lists the output of the hosts affected by the call.
`--list-tasks`	This provides an overview of the tasks of a playbook.
`--start-at-task` <NAME>	Start with task <NAME>; wildcards like * are possible.
`-i` <INVENTORY_FILE>	Use alternative inventory; see Chapter 2, Section 2.8.
`-v`	This indicates verbose mode (up to four repetitions are possible).
`-f` <NUM>	This indicates the number of forks; see Chapter 3, Section 3.4.1.

Table 5.2 "ansible-playbook": Most Important Options

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Options	Meanings
`--syntax-check`	This performs a syntax check of the playbook,
`--check`	This performs a simulated run (a dry run); see Chapter 10, Section 10.2.6.
`--diff`	This shows file changes, often with `--check`; see Chapter 10, Section 10.2.6.
`--step`	This indicates step-by-step execution.
`-t <TAGS>`	This means execute only tagged tasks; see Section 5.6.
`-e <VAR=VALUE>` / `-e @<file.yml>`	This indicates parameterization; see Chapter 6, Section 6.1.2.

Table 5.2 "ansible-playbook": Most Important Options (Cont.)

That should be sufficient for most applications. Please run the `ansible-playbook -h` command to get the complete overview.

5.8 An Example of an Apache Installation

Let's now turn for the first time to a more demanding yet still manageable task: an Apache installation with small adjustments to the homepage and its configuration. The goal here is to demonstrate certain typical approaches, rather than to provide a complete example of a real-life web server installation.

In addition, we limit ourselves to target hosts from the Debian family, as the Apache server is handled quite differently in our test distributions. Getting these differences (elegantly) under control will be a task for later.

For this example, you should name your playbook *apache.yml*. We'll work through the process task by task, and at the end, you will find the complete playbook as a whole.

5.8.1 Step by Step

You begin by clarifying the involved hosts and introducing the tasks section:

```
---
- hosts: [debian, ubuntu]

  tasks:
```

You'll see the two target hosts by using the more compact inline notation of a YAML list.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Let's start with the first task. On Debian-like systems, you should always work with current package lists when performing installations or upgrades. Debian users know the `apt update` and `apt-get update` commands for this, but in the Ansible context, you accomplish this with the `apt` module. To enable idempotency or to avoid constant unnecessary updates, you can use the `cache_valid_time` parameter as follows to specify a period in seconds during which the Apt cache is considered "fresh":

```
- name: Update package index
  apt:
    update_cache: yes
    cache_valid_time: 3600
```

You should also perform the actual installation with the `apt` module:

```
- name: Install Apache package
  apt:
    name: apache2
```

Starting the service and integrating it into the boot process are not really necessary on Debian systems because the package's post-install routines already handle that. However, you will probably soon have to deal with non-Debian systems, and in that case, you will definitely need to perform this step:

```
- name: Start service and enable start on boot
  service:
    name: apache2
    state: started
    enabled: yes
```

Finally, you should replace the default Apache welcome page with your own version. One way to upload files is by using the `copy` module. Typically, you would do this by specifying a source file on the control host with the `src` parameter, but here, you should use the direct inline notation with the `content` parameter for comprehensibility. You should also pay particular attention to the YAML block that's introduced with "|": which is a very elegant way to accommodate smaller text passages:

```
- name: Upload minimal homepage
  copy:
    dest: /var/www/html/index.html
    mode: 0644
    content: |
      <!doctype html>
      <html>
        <head>
          <meta charset="utf-8">
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
      <title>Test</title>
    </head>
    <body>
      <h1>Welcome!</h1>
    </body>
  </html>
```

5.8.2 The Complete Playbook

You can see the complete playbook in Listing 5.11:

```
---
- hosts: [debian, ubuntu]

  tasks:
    - name: Update package index
      apt:
        update_cache: yes
        cache_valid_time: 3600

    - name: Install Apache package
      apt:
        name: apache2

    - name: Start service and enable start on boot
      service:
        name: apache2
        state: started
        enabled: yes

    - name: Upload minimal homepage
      copy:
        dest: /var/www/html/index.html
        mode: 0644
        content: |
          <!doctype html>
          <html>
            <head>
              <meta charset="utf-8">
              <title>Test</title>
            </head>
            <body>
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
      <h1>Welcome!</h1>
    </body>
  </html>
```

Listing 5.11 "apache.yml": First Simple Playbook for Apache Installation

After you call `ansible-playbook apache.yml`, you should be able to admire the start pages in the browser. If your desktop system does not recognize the names of the Linux machines, you'll have to work with IP addresses like this one:

```
http://192.168.150.10
```

The `curl` tool is also a big help when you're doing more serious work in the HTTP environment,; please install it from the package of the same name if it's not already installed on your system. With that, you can quickly and easily retrieve content over HTTP without a browser (and you can even omit the `http://` prefix):

```
$ curl http://debian
<!doctype html>
[...]
    <h1>Welcome!</h1>
[...]
```

5.8.3 "--start-at-task, "--check," and "--diff"

Let's take another look at three useful options for `ansible-playbook`. Let's say you're currently developing the final task that uploads the content of the website. Even with our small playbook, it's tedious to have to wait for all preceding tasks to complete with each run.

Here, the `--start-at-task` option can be helpful because it lets you specify the entry point into the play itself. You need to specify the *name* of the task, but thankfully, the * wildcard is allowed, as in the following example:

```
$ ansible-playbook apache.yml --start-at-task "Upload*"

PLAY [debian,ubuntu] ***************************************************

TASK [Gathering Facts] *************************************************
ok: [debian]
ok: [ubuntu]

TASK [Upload minimal homepage] *****************************************
ok: [debian]
ok: [ubuntu]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
PLAY RECAP *********************************************************************
debian                     : ok=2    changed=0    unreachable=0    failed=0
ubuntu                     : ok=2    changed=0    unreachable=0    failed=0
```

You've probably noticed the idempotency at this point: if you don't make any substantial changes to the homepage, then there's also no change for Ansible. *However,* if you do make a change, then there is the interesting possibility of a dry run along with a display of the differences. This means Ansible can tell you what it *would do* without actually doing it. Suppose we change the headline of the homepage as follows:

```
$ ansible-playbook apache.yml --check --diff
[…]
--- before: /var/www/html/index.html
+++ after: /home/vagrant/.ansible/tmp/ansible-local-13801htx_o2m6/tmpeyumjvg1
@@ -5,6 +5,6 @@
     <title>Test</title>
   </head>
   <body>
-    <h1>Welcome!</h1>
+    <h1>Welcome to our homepage!</h1>
   </body>
 </html>

changed: [debian]

[…]
```

You can see how the content of the `<h1>` line would change. Of course, on your system, the output is colorful and therefore much more understandable.

5.9 Handler: Executing Tasks Only on Changes

Now, we'll extend our Apache playbook a bit. The Apache server should receive a small configuration adjustment in the form of a plug-in configuration file.

5.9.1 Step by Step Execution

Let's take a redirect as an example (for this, you usually don't need to include another Apache module):

```
    - name: Upload config file
      copy:
        dest: /etc/apache2/conf-available/redirect.conf
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
      content: |
        Redirect /go https://www.google.com
```

However, on operating systems from the Debian family, this is not enough; you must also include or activate the new configuration. You can use the `a2enconf` command for this purpose; it creates a symbolic link in the sibling *conf-enabled* directory. You can take advantage of this by using the `creates` parameter to ensure idempotency:

```
  - name: Activate config
    command:
      cmd: a2enconf redirect
      creates: /etc/apache2/conf-enabled/redirect.conf
```

What is happening here? The command to be executed is `a2enconf redirect`, and it's specified by `cmd`. If that were all, Ansible would execute this command in every run and always report a change, because commands are never idempotent by default. In this case, that wouldn't be bad—it just wouldn't be nice. To achieve idempotency, you can specify a file with the `creates` addition, and if that file already exists, then Ansible will simply skip executing the command and will report "ok." This fits perfectly in our case because the command actually creates a file (or more precisely, a symbolic link) upon successfully executing the command. In other situations, you might need to handle `creates` more critically because it always concerns the *existence* of a file and not whether *the right content is inside*.

Finally, you need to reload the Apache service, as follows:

```
  - name: Reload Apache service
    service:
      name: apache2
      state: reloaded
```

Then, you run the playbook with the new steps and then test it in the browser or simply by using `curl` again:

```
$ curl http://debian/go
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="https://www.google.com">here</a>.</p>
<hr>
<address>Apache/2.4.62 (Debian) Server at debian Port 80</address>
</body></html>
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

So far, everything looks fine, and the only slightly annoying part is the fact that the `reloaded` and `restarted` states of the `service` module are *never idempotent*. In other words, the reload is always performed, regardless of whether a configuration change has taken place or not—and this is where *handlers* prove useful.

5.9.2 Handlers

Handlers are an elegant solution to the problem of unnecessary reloads/restarts, which are only called when a change has actually occurred. For this, you notify the handler with `notify` in the task that potentially makes the change:

```
  - name: Upload config file
    copy:
      dest: /etc/apache2/conf-available/redirect.conf
      content: |
        Redirect /go https://www.google.com
    notify: reload apache
```

Again, you should pay attention to the indentation: `notify` is at the level of the task, and it is not a copy parameter! The value of `notify` is in the form of text; preferably, you should choose a short, technical, and clearly understandable description.

The actual reload task now moves into a new play section called `handlers`, which starts at the same level as `tasks`. You can establish the connection between the two tasks by adding a `listen` directive that has the same text value as the above `notify`:

```
  handlers:
   - name: Reload Apache service
     service:
       name: apache2
       state: reloaded
     listen: reload apache
```

If you test the whole thing now, everything should work as desired, with no change and no reload. However, if you modify the redirect as a test, then you'll get the following result:

```
$ ansible-playbook apache.yml
[…]
TASK [Upload config file] ******************************************************
changed: [debian]
changed: [ubuntu]
[…]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
RUNNING HANDLER [Reload Apache service] ************************************
changed: [debian]
changed: [ubuntu]

PLAY RECAP *********************************************************************
debian                     : ok=8    changed=2    unreachable=0    failed=0
ubuntu                     : ok=8    changed=2    unreachable=0    failed=0
[…]
```

Explanations

- A task can notify one or more handlers by using `notify`. If they have a matching `listen`, then they are called in the event of a change.
- If multiple handlers need to be notified, then you can either pass a list to `notify` or equip all relevant handlers with the same `listen`.
- Handlers always run (if at all) at the end of the play.
- Handlers run in exactly the order in which they are specified under `handlers`.
- *If a play fails after a* `notify` *has already occurred, then the handlers do not run by default!*

The last point is really noteworthy (and actually *not* desirable). However, you can change this default behavior in three different ways:

1. With the `--force-handlers` command line switch
2. With the `force_handlers: true` setting in the playbook
3. With the `force_handlers = true` setting in the *ansible.cfg*, as follows (we recommend this approach):

```
# [defaults]
force_handlers = true
```

Listing 5.12 "ansible.cfg" Excerpt for Mandatory Invocation of Notified Handlers

Invoking Handlers Earlier

Quite often, you'll need to trigger certain handlers not only at the end of the play but earlier. If necessary, you can force the invocation of all *previously notified handlers* as follows:

```
    # [... tasks ...]

    - meta: flush_handlers

    # [... tasks ...]
```

Listing 5.13 Triggering Handlers Earlier

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The Simpler Form without "listen"

You can also create a handler without `listen`. In this case, the `name` field of the handler must match the `notify` field of the task:

```
tasks:
  - name: Upload config file
    copy:
      dest: /etc/apache2/conf-available/redirect.conf
      content: |
        Redirect /go https://www.google.com
    notify: reload apache

handlers:
  - name: reload apache
    service:
      name: apache2
      state: reloaded
```

Listing 5.14 Classic Handler Linkage without "listen"

There is nothing fundamentally wrong with this approach, and people still often use it—perhaps because it is somewhat shorter. However, in my opinion, the variant with `listen` has two advantages:

- The `name` field of a handler can continue to fulfill its descriptive purpose—preferably with a "lengthy" text. Such texts would be rather inconvenient to use in a `notify`.
- You can equip different handlers with the same `listen`, as demonstrated by the following example from the Ansible documentation:

```
tasks:
  - name: restart everything
    command: echo "this task will restart the web services"
    notify: "restart web services"

handlers:
  - name: restart memcached
    service: name=memcached state=restarted
    listen: "restart web services"

  - name: restart apache
    service: name=apache state=restarted
    listen: "restart web services"
```

Listing 5.15 Different Handlers Listening to Same "notify"

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

5.9.3 The Complete Playbook up to This Point

Finally, you should name the new version *apache2.yml* (which is hopefully not misleading):

```
---
- hosts: [debian, ubuntu]

  tasks:
    - name: Update package index
      apt:
        update_cache: yes
        cache_valid_time: 3600

    - name: Install Apache package
      apt:
        name: apache2

    - name: Start service and enable start on boot
      service:
        name: apache2
        state: started
        enabled: yes

    - name: Upload minimal homepage
      copy:
        dest: /var/www/html/index.html
        mode: 0644
        content: |
          <!doctype html>
          <html>
            <head>
              <meta charset="utf-8">
              <title>Test</title>
            </head>
            <body>
              <h1>Welcome!</h1>
            </body>
          </html>

    - name: Upload config file
      copy:
        dest: /etc/apache2/conf-available/redirect.conf
        content: |
          Redirect /go https://www.google.com
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
      notify: reload apache

    - name: Activate config
      command:
        cmd: a2enconf redirect
        creates: /etc/apache2/conf-enabled/redirect.conf


  handlers:
    - name: Reload Apache service
      service:
        name: apache2
        state: reloaded
      listen: reload apache
```

Listing 5.16 "apache2.yml": Playbook with Handler

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 6
Playbooks and Tasks: Advanced Methods

The central theme of this chapter is the handling of *variables*. Almost no real-world playbook can do without them, because only with variables can you centralize settings or values or handle differences between target systems efficiently.

Other points we discuss in this chapter include handling Jinja templates and control structures such as conditions and loops.

6.1 Variables

You can access variables in Ansible tasks, but to do this, you need to define such variables beforehand.

There are countless ways to define variables (more precisely, there are over 20). In the following sections, we will look at some of these possibilities.

6.1.1 Play Vars

First off, you can define variables in the `vars` section in a play. In the example in Listing 6.1, you can see how two variables are defined and their contents are then output in the only task using the `debug` module:

```
---
- hosts: localhost
  gather_facts: false

  vars:
    color: blue
    number: 42

  tasks:
    - debug:
        msg: "Color: {{color}}, number: {{number}}"
```

Listing 6.1 "vars1.yml": First Example of Defining Variables in Playbook

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
$ ansible-playbook vars1.yml
[…]
ok: [localhost] => {
    "msg": "Color: blue, number: 42"
}
[…]
```

No surprises so far. You just need to accept the double curly braces (`'{{...}}'`) as the syntax required for variable access—which may be unfamiliar, but it's quite logical.

6.1.2 Extra Vars

Another way to define variables is to pass them in the command line (Ansible calls these *extra vars*). For demonstration purposes, we'll use the same example playbook as in the last section:

```
$ ansible-playbook vars1.yml -e number=100
[…]
ok: [localhost] => {
    "msg": "Color: blue, number: 100"
}
[…]
```

Passing via command line overrides the definition in the play—again, no surprise. You can repeat the `-e` switch as often as needed:

```
$ ansible-playbook vars1.yml -e color=red -e number=100
```

Alternatively or additionally, you can specify a file that contains value assignments in YAML map syntax with `-e @<file.yml>`:

```
color:  orange
number: 200
```

Listing 6.2 "values.yml"

A corresponding call would look like this:

```
$ ansible-playbook vars1.yml -e @values.yml
```

6.1.3 Precedence

If you define the same variable in multiple places, then one of them will take precedence. For example, as we have seen, an extra vars variable is stronger than a play variable.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Of course, the precedence rules are clearly defined, and you can find the detailed ranking of priorities at *https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html* in the "Variable precedence: Where should I put a variable?" section.

I've also included the ranking of priorities here. Note that the order of the list progresses from *weak to strong*, meaning that what is further down wins.

```
 1. command line values (e.g., "-u user")
 2. role defaults
 3. inventory file or script group vars
 4. inventory group_vars/all
 5. playbook group_vars/all
 6. inventory group_vars/*
 7. playbook group_vars/*
 8. inventory file or script host vars
 9. inventory host_vars/*
10. playbook host_vars/*
11. host facts / cached set_facts
12. play vars
13. play vars_prompt
14. play vars_files
15. role vars (defined in role/vars/main.yml)
16. block vars (only for tasks in block)
17. task vars (only for the task)
18. include_vars
19. set_facts / registered vars
20. role (and include_role) params
21. include params
22. extra vars (always win precedence)
```

As you can see, play vars ranks twelfth on the list and extra vars ranks twenty-second—which proves that extra vars on the command line always win!

6.1.4 "set_fact"

Sometimes, you'll need to define variables only during a play's runtime. Using `set_fact`, you can introduce variables as needed or assign new values to existing variables:

```
---
- hosts: localhost
  gather_facts: false

  tasks:
    - name: Define variables
      set_fact:
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
        color: blue
        number:  42

    - debug:
        msg: "Color: {{color}}, number: {{number}}"
```

Listing 6.3 "set_fact.yml": Variable Definition with "set_fact"

On the precedence list, `set_fact` ranks nineteenth (so it's quite strong).

Note

Perhaps the unusual name `set_fact` still needs some explanation. Wouldn't something like `set_variable` have been more plausible?

Facts, which you will soon learn about in Section 6.3, are host-specific variables in Ansible. The same applies to variables that you define using `set_fact`: they are only visible to the host that executed the `set_fact` task from the moment of definition.

So, this name was probably chosen to clearly indicate the conceptual relationship to these facts.

6.1.5 "group_vars"

To handle differences between systems, you need methods that allow a variable to have different values depending on the system. For this, you first take all hosts on board and then remove all variable definitions:

```
---
- hosts: all
  gather_facts: false

  tasks:
    - debug:
        msg: "Color: {{color}}, number: {{number}}"
```

Listing 6.4 "vars2.yml": Playbook Dependent on External Parameterization

This also allows you to directly verify that accessing an undefined variable ends with a fatal runtime error, as follows:

```
$ ansible-playbook vars2.yml

TASK [debug] *******************************************************************
fatal: [ubuntu]: FAILED! => {"msg": "The task includes an option with an
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
undefined variable. The error was: 'color' is undefined. […]

[… the same for all other hosts …]
```

How can we now provide variables outside of a playbook (apart from the extra vars)?

One of the places Ansible searches is a directory named `group_vars`. It does this in one of two ways:

- In addition to the used inventory
- In addition to the used playbook

In this directory, Ansible expects to find one of two things:

- A *<GROUPNAME>.yml* file
- A *<GROUPNAME>/* directory with arbitrarily named **.yml* files

Now, you should try this by creating a *group_vars* folder. Whether you create it alongside the inventory or in the *playbooks/* folder is a technical question: are the parameters tied to the hosts, or are they dependent on certain playbooks? In our play example, it doesn't matter, so you should try the first variant:

```
$ mkdir ~/ansible/projects/start/group_vars
```

Now, let's take advantage of the fact that the `all` group contains every host and place the following *all.yml* file in *group_vars/*, which thus provides default values for all hosts:

```
---
color: lightgray
number: 50
```

Listing 6.5 "group_vars/all.yml": Default Values for All Hosts

Now, our playbook should be able to run successfully, as follows:

```
$ ansible-playbook vars2.yml
[…]
ok: [debian] => {
    "msg": "Color: lightgray, number: 50"
}
[…]
```

Although we still don't have individual parameterization, we could now easily provide additional files for more specific groups. For example, suppose you want to "color" all SUSE hosts "green." To do this, you could define a `suse_hosts` group in the inventory, as follows:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
# ...
[suse_hosts]
suse
# ...
```

Listing 6.6 "inventory", Inventory Section with Additional Group

This would also cause Ansible to consider and read the following *group_vars/suse_hosts.yml* file as a parameter file:

```
---
color: green
```

Listing 6.7 "group_vars/suse_hosts.yml": Separate Configuration for SUSE Hosts

And thus, it results in this:

```
$ ansible-playbook vars2.yml
[…]
ok: [ubuntu] => {
    "msg": "Color: lightgray, number: 50"
}
ok: [suse] => {
    "msg": "Color: green, number: 50"
}
[…]
```

Let me reiterate the precedence that applies here: the *group_vars/all.yml* file ranks fourth on the list (very weak) and a more specific *group_vars/*.yml* file ranks sixth on the list (somewhat stronger).

6.1.6 "host_vars"

You can probably already guess how it continues here: Ansible also supports individual parameterization down to the level of individual hosts. All concepts work analogously to `group_vars`, but I will still summarize.

Ansible also searches in a directory named `host_vars` in one of two ways:

- In addition to the used inventory
- In addition to the used playbook

In this directory, Ansible then expects one of the following two things:

- A *<HOSTNAME>.yml* file
- A *<HOSTNAME>/* directory with arbitrarily named *.yml files therein

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

As you'd expect, individual host parameters should be stronger than group parameters, and a quick look at the precedence list confirms that: this method is ranked ninth.

6.1.7 "vars_files": Variables in Any External Files

In addition to `host_vars` and `group_vars` files, you can maintain variables in any external files, which you can then include in the play by using `vars_files`:

```
- hosts: all

  vars_files:
    - vars/setup.yml
    - [ vars/additional.yml, vars/common.yml ]
```

Here, Ansible would load the specified files in order, with the *vars/setup.yml* file required to be present (otherwise, the play would abort). If, as in the second line, a list of files is specified, then Ansible will try to find and load one of them in sequence. The first existing file wins—but if Ansible doesn't find one, then the play will also abort here with an error. This method ranks fourteenth on the list.

6.1.8 Prompting

We don't often use the (not uninteresting) option to query variables as user input at the start of runtime by using a `vars_prompt` section:

```
---
- hosts: localhost
  gather_facts: false

  vars_prompt:
    - name: var1
      prompt: Please enter value for var1
      default: 42
      private: false

    - name: var2
      prompt: And another value for var2 (no one should see this)
      private: true

  tasks:
    - debug:
        msg: "var1 is {{var1}}, var2 is {{var2}}"
```

Listing 6.8 "prompting.yml": Interactive Variable Input during Runtime

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
$ ansible-playbook prompting.yml
Please enter value for var1 [42]: 100
And another value for var2 (no one should see this): ______
[…]
ok: [localhost] => {
    "msg": "var1 is 100, var2 is secret"
}
[…]
```

A call with corresponding extra vars would bypass the prompting. (The somewhat tricky quoting is only necessary because of the included space.):

```
$ ansible-playbook -e var1=111 -e var2="'Some text'" prompting.yml
```

This method ranks thirteenth on the list.

6.1.9 Accessing Complex Structures

Variables can contain not only simple single values but also complex nested structures, as the following example shows:

```
---
- hosts: localhost
  gather_facts: no

  vars:
    data:
      beginning: Hello!
      end: Best regards
      lines:
        - How are you doing?
        - The weather is very nice.

  tasks:
    - name: Upload /tmp/message.txt
      copy:
        dest: /tmp/message.txt
        content: |
          {{ data.beginning }}

          {{data.lines[0]}}
          {{data.lines[1]}}

          {{ data.end }}
```

Listing 6.9 "vars-complex.yml": Working with Complex Structured Variables

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Here are some explanations:

- To access values in nested maps, use the following notation:

  `mapname.key1.key2…`

  Alternatively, you can use this syntax:

  `mapname['key1']['key2'][…]`

  This alternative syntax becomes indispensable if the name of a key is to be taken from another variable.
- You can access list values with the following notation:

  `listname[POS]`

  Indexing starts as usual at 0.

6.1.10 "assert" Tests

In larger environments, you'll quickly find yourself in a situation where you're dealing with both playbook authors and playbook users. But how can a playbook user know which variables are actually needed? Should they just run the playbook and be disappointed to find that it crashes halfway through?

First and foremost, appropriate documentation would be helpful here, but unfortunately, it is not always available in practice. One way to improve the situation is to use the `assert` module in a very early task to check for the presence of all required values, as follows:

```
---
- hosts: debian
  gather_facts: no

  tasks:
    - assert:
        that:
          - hostname is defined and hostname != ''
        fail_msg: The variable hostname is not defined or empty

    - name: Set hostname
      hostname:
        name: "{{ hostname }}"
```

Listing 6.10 "assert.yml": Ensuring Certain Requirements

All conditions of `that` must be met or the play will stop at this point. In this example, there is only one condition, which makes sense if you want to optionally work with a

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

customized error message (`fail_msg`). (If there are multiple conditions, then of course, the error message cannot know which one may have failed.)

We'll talk about the syntax of these conditions soon. When you only need to read conditions (as in this case), you'll usually have no problems understanding them.

6.2 Registered Variables

Especially in the early phase, when you start to familiarize yourself with Ansible, you'll often call Linux commands with `command` or `shell`—and later, you'll realize that using a specific module would have been better. You could have used the `file` module instead of a `mkdir` call, the `user` module instead of a `useradd` call, the `synchronize` module instead of a `rsync` call.

But first, you would have to know all of this, and second, there isn't a module for everything. So now, we should take a look at how `command` and `shell` calls behave in the playbook context. (You already know these as ad hoc calls; see Chapter 3, Section 3.1.) Let's start again with a simple example:

```
---
- hosts: debian
  gather_facts: no

  tasks:
    - name: Determine space requirements
      command: df -h /
```

Listing 6.11 "command1.yml", Execution of Linux Command

```
$ ansible-playbook command1.yml

PLAY [debian] ******************************************************************

TASK [Determine space requirements] ********************************************
changed: [debian]

PLAY RECAP *********************************************************************
debian                     : ok=1    changed=1    unreachable=0    failed=0
```

You'll notice immediately that, in contrast to an ad hoc call, you see nothing. Upon closer inspection, you'll also see that Ansible reports a change (even though we are quite sure that a `df -h` should not cause any changes to the target system). The latter is easy to explain: Ansible does not perform a deep analysis of the Linux commands that are about to be executed, and therefore, it simply does not know the effect here. By default, Ansible always assumes a change for such a task—but you can define this

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

assessment yourself with the task `changed_when` attribute. Once you've finished learning about conditions in Ansible (in Section 6.4), you'll have many options available. However, in this case, simple negation using `changed_when: false` is already sufficient. A `df` command never causes a change to the system:

```
---
- hosts: debian
  gather_facts: no

  tasks:
    - name: Determine space requirements
      command: df -h /
      changed_when: false
```

Listing 6.12 "command2.yml": Calling Linux Command without Change

However, the issue remains that you do not see the output of the command.

First of all, you should come to terms with the fact that playbooks are not meant to produce pretty output. Rather, they are intended to perform actions on target systems to achieve a desired target state, and pretty output is irrelevant.

Therefore, all standard outputs from `command` and `shell` calls are suppressed. However, we can access them (and many other pieces of information) by taking a small detour: Ansible can store all information about the execution of a task in a structured variable (the `register` keyword). You can then access that variable in later tasks, as follows:

```
---
- hosts: debian
  gather_facts: no

  tasks:
    - name: Determine space requirements
      command: df -h /
      changed_when: false
      register: df_cmd

    - debug:
        msg: '{{df_cmd.stdout_lines}}'
```

Listing 6.13 "command3.yml": Execution of Linux Command Followed by Output

```
$ ansible-playbook command3.yml

PLAY [debian] ******************************************************************

TASK [Determine space requirements] ********************************************
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
ok: [debian]

TASK [debug] ********************************************************************
ok: [debian] => {
    "msg": [
        "Filesystem                         Size  Used Avail Use% Mounted on”:
        "/dev/mapper/debian--12--vg-root   62G  1.3G   57G   3% /"
    ]
}

PLAY RECAP **********************************************************************
debian                     : ok=2    changed=0    unreachable=0    failed=0
```

First, I would like to clarify the technical classification of registered variables:

Important

Registered variables are technically the same as `set_fact` variables. They are *host specific*, meaning they are only visible to the host that has just executed the task. They also have exactly the same precedence.

A registered variable has various fields or attributes that depend on the type of task. Here, we accessed the `stdout_lines` field. It contains a list of the individual output lines, so it is close what would have been printed. (You can somewhat influence the display of such structures on the screen; see Chapter 10, Section 10.2.2 for more information.)

Table 6.1 shows all the fields that would appear if you were to output the complete `df_cmd` variable in the `debug` task:

Attributes	Values
`changed`	Did the task make a change to the system?
`failed`	Did the task fail?
`cmd`	This is the command with parameters.
`rc`	This is the UNIX return code of the command.
`start`	This is the start time.
`end`	This is the end time.
`delta`	This is the execution duration.
`stdout`	This is the joined standard output with newlines as `\n`.
`stdout_lines`	This is the list of individual standard output lines.

Table 6.1 Attributes of “command” or “shell” Task

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Attributes	Values
stderr	This is the standard error output that's similar to stdout.
stderr_lines	This is the standard error output that's similar to stdout_lines.

Table 6.1 Attributes of "command" or "shell" Task (Cont.)

Tip

The debug module offers an even simpler way to output the contents of a variable. Instead of using

```
- debug: msg='{{some_variable}}'
```

you can simply use

```
- debug: var=some_variable.
```

6.3 Facts and Implicit Variables

This section is about two other very important types of host-specific variables: the so-called *facts* and the *implicit variables*.

6.3.1 Facts

If you set gather_facts to true/yes in a playbook or omit the line altogether, Ansible will gather information about the target host(s) at the start. The gathered information will then be available in the ansible_facts data structure, and you will be able to use it just like normal variables.

As an experiment, you should run it once for any host:

```
$ ansible <HOST> -m setup
```

Take a look at the whole thing at your leisure. (For example, the Debian host from our lab environment produces over five hundred lines of text, and depending on the system and environment, it can sometimes be over seven hundred—but we will spare ourselves from printing them here.)

In ad hoc mode, you can also filter if you don't want to see all facts. For instance, you can filter on all facts that contain distribution information, as follows:

```
$ ansible debian -m setup -a 'filter=ansible_distribution*'
debian | SUCCESS => {
    "ansible_facts": {
        "ansible_distribution": "Debian",
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
        "ansible_distribution_file_parsed": true,
        "ansible_distribution_file_path": "/etc/os-release”:
        "ansible_distribution_file_variety": "Debian",
        "ansible_distribution_major_version": "12",
        "ansible_distribution_minor_version": "6",
        "ansible_distribution_release": "bookworm",
        "ansible_distribution_version": "12"
    },
    "changed": false
}
```

Unfortunately, the output of the `setup` module is somewhat misleading. For example, if you look at the upper section, you will find the `ansible_architecture` key with a simple value like `x86_64`:

```
$ ansible -m setup debian
debian | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.2.15”:
            "192.168.150.10"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fedb:4145",
            "fe80::a00:27ff:fef2:77c0"
        ],
        "ansible_apparmor": {
            "status": "enabled"
        },
        "ansible_architecture": "x86_64",
[…]
```

If you want to use this value in a playbook, then based on everything you learned in Section 6.1.9, you would write something like this:

```
    - debug: var=ansible_facts.ansible_architecture
```

Unfortunately, you'd only get "`Undefined`" as a value in return. However, you would be successful with this:

```
    - debug: var=ansible_facts.architecture

    # or(!)

    - debug: var=ansible_architecture
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Let's note the following:

- When you access facts via the `ansible_facts` map, you must always use the first-level keys *without* the `ansible_` prefix.
- The first-level keys are also available as direct variables *with* the `ansible_` prefix.

You can also enable the latter option via configuration, specifically with the `inject_facts_as_vars` setting. This setting was introduced with Ansible 2.5 and is set to `true` by default.

6.3.2 Caching of Facts

Gathering facts on a host can be time consuming and performance intensive, but you can cache the result for a period you can define—in simple JSON or YAML files or in a NoSQL database (which will often be Redis). You can achieve caching with JSON files (which is sufficient as long as you do not want to set up a central cache) with the following configuration,:

```
# [defaults]
gathering = smart
fact_caching = jsonfile
fact_caching_connection = ~/.ansible/fact_cache
fact_caching_timeout = 86400
```

Listing 6.14 "ansible.cfg": Excerpt for Caching Facts

With a `fact_caching_connection`, you can choose any directory, which will be automatically created if necessary. Of course, you can freely choose the cache expiration time in seconds by using `fact_caching_timeout`. (In this case, it's 86,400 seconds = 1 day.)

To trigger a renewal of the cache, you can do the following:

1. Manually delete the cache files.
2. Invoke the `setup` module as an ad hoc command or playbook task.
3. Call `ansible-playbook --flush-cache <playbook.yml>`.

Warning: Everything Is Cached!

With an active facts cache, all facts are cached—even those it makes little sense to cache (at least at first glance), such as `ansible_date_time`.

There has been ample discussion of this in the Ansible community, which has ultimately decided that would be too cumbersome to exclude individual facts from caching. Ansible developers have also noted that `ansible_date_time` does not exactly match the current server time anyway, as fact gathering always happens at the beginning of the playbook run, even without caching.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Other users have also mentioned that they use `ansible_date_time` to determine how old the cache content is—and that they would therefore be very annoyed if the usual behavior were to change.

6.3.3 Implicit Variables

In Ansible, there are also *implicit variables* that are always present. Table 6.2 shows a selection of the most important and interesting ones:

Variables	Meanings
`inventory_hostname`	This is the name of the host in the inventory.
`inventory_hostname_short`	This is the inventory hostname without the domain.
`group_names`	This is the list of all groups the current host is a member of.
`groups`	This is the ,ap with all inventory groups and their contents.
`hostvars`	This is the map with all host variables of all inventory hosts (including facts).
`ansible_play_hosts_all`	These are all hosts of the current play.
`ansible_play_hosts`	These are all hosts of the current play (excluding failed and unreachable hosts).
`playbook_dir`	This is the directory where the current playbook is located.
`inventory_dir`, `inventory_file`	This is the directory or file of the current inventory host.
`role_path`, `role_name`	This is the file or name of the current role (more on this later).
`ansible_version`	This is the version information on the Ansible that's in use.
`ansible_check_mode`	This indicates whether the check mode is enabled or not.
`ansible_limit`	This is the content of the command line switch `-l` or `--limit`.

Table 6.2 Implicit Ansible Variables

You can find a complete overview of these variables at *https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html*; there, implicit variables are referred to as *magic variables*.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

6.3.4 Example

When you use a variable, it doesn't matter what kind of variable it is. In Listing 6.15, you can see an example in which a lot of typical facts and implicit variables are used:

```
---
- hosts: all

  tasks:
    - name: Some facts
      debug:
        msg: >-
          {{inventory_hostname}} ({{ansible_host}})
          is a {{ansible_distribution}},
          version {{ansible_distribution_major_version}}
          ({{ansible_distribution_version}}),
          family {{ansible_os_family}}

    - name: Some magic variables
      debug:
        msg: >-
          Your playbook is in {{playbook_dir}},
          your inventory in {{inventory_dir}}.
      run_once: true

    - name: Group information
      debug:
        msg: Your groups (with hosts) are {{groups}}
      run_once: true

    - name: Host information
      debug:
        msg: Your Hosts are {{groups.all | sort}}
      run_once: true

    - name: Hostinfo
      debug:
        msg: >-
          IPv4-addresses of {{item}} are
          {{hostvars[item].ansible_all_ipv4_addresses}}
      with_items: '{{groups.all | sort}}'
      run_once: true
```

Listing 6.15 "facts-and-magic-vars.yml": Example of Use of Commonly Used Facts and Implicit Variables

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

The `run_once` additions ensure that the respective tasks run only once instead of repeatedly for each host. (Running them repeatedly would be pointless because they are always the same.) The last task is a loop over all inventory hosts, and I will discuss this technique in a bit.

Note

Pay attention to the *family affiliation* (`ansible_os_family`) of our Rocky Linux. The correct value here is `RedHat` because Rocky is nothing more than an RHEL clone (just like CentOS used to be).

In older versions of Ansible, such as the one included with Debian 11, this fact was not set correctly; there, you would see `Rocky` instead of `RedHat`. If you had worked with this fact at that time, it could have caused problems after an Ansible upgrade.

We tell you this for no particular reason . . . except perhaps to remind you that a newer Ansible is almost always better than an older Ansible.

6.3.5 External Information Gatherers: "facter" and "ohai"

If the `facter` and `ohai` commands are installed on a target host, then they are *automatically additionally* called during information gathering.

The corresponding facts are then included in the Ansible facts with the `facter_` or `ohai_` prefix.

6.3.6 Not Enough Facts Yet? Try "/etc/ansible/facts.d"!

If all this is not enough for you, then you can provide your own facts.

You do this by creating a directory */etc/ansible/facts.d/* on a target host and placing **.fact* files in JSON format there, as follows:

```
{
  "os_coolness_factor": 9,
  "preferred_coffee_strength": "strong"
}
```

Listing 6.16 "/etc/ansible/facts.d/test.fact": Example Facts File

These additional facts are then included in `ansible_local` in the facts of the corresponding target host.

Instead of JSON, you can also use the somewhat less flexible but simpler INI format. It may interest you that you can also place *executable programs* there and that they write the desired result to `STDOUT`!

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

6.4 Conditional Execution with “when”

In Ansible, you can associate tasks with a condition, which means they will only be executed when some condition is met. You do this by using the `when` task attribute. You can include all types of variables and facts, as the following example shows:

```
- name: Output only on Debian-like systems
  debug: msg="Hello Debian or Debian-like system!"
  when: ansible_os_family == "Debian"
```

As usual, you can combine conditions with `not`, `and`, and `or`—and of course, you can use parenthesis like this `(...)` to clarify precedence or increase comprehensibility.

Note

Technically, a `when:` condition is evaluated as a Jinja expression, which we will discuss in Section 6.6. At the moment, it’s best to simply look at the following examples and hopefully agree with me that the syntax is quite “plausible.”

As in SQL, there is an `in` operator that you can use to check for occurrences in lists:

```
when: ansible_os_family in ["Debian": "Suse"]
```

The syntax becomes a bit more subtle when the element to be checked is a text literal. In that case, you must double-quote or use a YAML block expression:

```
when: '"Cat" in list_of_animals'

when: >
  "Cat" in list_of_animals
```

If you want to test whether a variable has a defined value or not, you can use constructs like this:

```
when: myvar is defined
when: myvar is undefined
```

A common idiom is testing whether a variable is defined *and* has a specific value. You can do this either classically or with some Jinja magic:

```
when: myvar is defined and myvar == "whatever"
when: myvar | default("") == "whatever"
```

Sometimes, you have to compare software version numbers, but the simple approach often fails because what looks like numbers is in fact internally represented as a text:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    # THIS DOES NOT WORK:
    [...]
    when: ansible_distribution_version >= 9
```

And, if you think of versions like '19.10' or even '4.19.0', then a numerical comparison becomes very questionable. Ansible offers the `version` test for this, as follows:

```
---
- hosts: all

  tasks:
    - debug: msg="A fairly new Ubuntu"
      when: ansible_distribution == "Ubuntu"
              and
            ansible_distribution_version is version('23.10', '>=')
```

Listing 6.17 "version-compare.yml": Comparison of Version Numbers

For more options, please refer to these pages:

- *https://docs.ansible.com/ansible/latest/user_guide/playbooks_conditionals.html*
- *https://docs.ansible.com/ansible/latest/user_guide/playbooks_tests.html*

If you've already experimented a bit with `when` conditions, then you've surely noticed the `skipping` lines in the output of `ansible-playbook`. Of course, this is completely plausible: whenever the condition *is not* met for a target hosts, such a line is shown:

```
TASK [Example] ************************************************************
ok: [debian]
skipping: [rocky]
skipping: [rocky]
skipping: [suse]
```

If you find all these `skipping` lines more annoying than informative, you can suppress them with the `display_skipped_hosts` configuration setting:

```
# [defaults]
display_skipped_hosts = false
```

Listing 6.18 "ansible.cfg": Code Excerpt for Suppressing Skip Outputs

Alternatively, you can use the `ANSIBLE_DISPLAY_SKIPPED_HOSTS` environment variable.

6.5 Handling System Differences: What's the Deal Now?

If you remember our Apache example, you'll remember that the last case (in Chapter 5, Section 5.9.3) was a very well-functioning variant for Debian and Ubuntu systems.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Meanwhile, you have a wealth of options at hand to extend the installation to all hosts (including `suse` and `rocky`)—but how, exactly?

The first problem is that the Apache package doesn't have the same name on all distributions. Let's consider various possibilities based on this issue.

If you need the right answer as quickly as possible, then all methods except the clumsy one will be good, and ultimately, which method you use will again be a matter of taste. In most cases, I would recommend the modular approach and the solid method only if you don't want more external files.

Attention, Firewall!

In all the Vagrant images used in our lab, the firewall is disabled. However, if you use any other standard installation of SUSE or Rocky Linux, then a packet filter is most likely active. You could, of course, manually disable it on the respective test systems as follows:

```
# systemctl disable --now firewalld
```

You can also do it in style with Ansible, which is a nice exercise. However, if you ever want to do it "the right way" (i.e., by leaving the firewall enabled and opening port 80 or 443), you will find appropriate tips for that in Chapter 19, Section 19.2.3.

6.5.1 The Clumsy Method

The clumsy method of compensating for system differences is as follows:

```
---
- hosts: all

  tasks:
    - name: Install Apache on Debian/Ubuntu systems
      apt:
        name: apache2
      when: ansible_os_family == "Debian"

    - name: Install Apache on Rocky systems
      dnf:
        name: httpd
      when: ansible_distribution == "Rocky"

    - name: Install Apache on SUSE systems
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    zypper:
      name: apache2
    when: ansible_distribution == "openSUSE Leap"
```

Listing 6.19 "apache-method-1.yml": Clumsy Method of Compensating for System Differences

This works fine, but it's about as elegant as the proverbial bull in a china shop. You'd only want to use it if you needed to have these distinctions in almost every task from now on: the package must be named differently, the service must be named differently, and the homepage and the configuration file must be copied to a different location. In most cases . . . . no thank you.

6.5.2 The Solid Method

The solid method looks like this:

```
---
- hosts: all

  tasks:
    - name: Debian/Ubuntu parameters
      set_fact:
        apache_package_name: apache2
      when: ansible_os_family == "Debian"

    - name: Rocky parameters
      set_fact:
        apache_package_name: httpd
      when: ansible_distribution == "Rocky"

    - name: SUSE parameters
      set_fact:
        apache_package_name: apache2
      when: ansible_distribution == "openSUSE Leap"


    - name: Install Apache package
      package:
        name: "{{apache_package_name}}"
```

Listing 6.20 "apache-method-2.yml": Solid Method of Balancing System Differences

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Here, a variable is set depending on the system, which contains the package name. Thanks to the generic `package` package module, only one task is needed for the actual installation.

At first glance, this method seems a bit more complex—but once you realize that in the first three tasks, you can parameterize *all* system-dependent properties and then only access them in the actual work tasks, the advantages of this method should become clear. Let's actually extend this to the first two work tasks of the playbook:

```
---
- hosts: all

  tasks:
    - name: Debian/Ubuntu parameters
      set_fact:
        apache_package_name: apache2
        apache_service_name: apache2
      when: ansible_os_family == "Debian"

    - name: Rocky parameters
      set_fact:
        apache_package_name: httpd
        apache_service_name: httpd
      when: ansible_distribution == "Rocky"

    - name: SUSE parameters
      set_fact:
        apache_package_name: apache2
        apache_service_name: apache2
      when: ansible_distribution == "openSUSE Leap"

    - name: Install Apache package
      package:
        name: "{{apache_package_name}}"

    - name: Start service and enable start on boot
      service:
        name: "{{apache_service_name}}"
        state: started
        enabled: yes
```

Listing 6.21 "apache-method-2b.yml": Solid Method with Two Tasks

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

6.5.3 The Tricky Method

The tricky method is as follows:

```
---
- hosts: all

  vars:
    apache:
      Debian:
        package_name: apache2
        service_name: apache2
      Ubuntu:
        package_name: apache2
        service_name: apache2
      Rocky:
        package_name: httpd
        service_name: httpd
      openSUSE Leap:
        package_name: apache2
        service_name: apache2

  tasks:
    - name: Install Apache package
      package:
        name: "{{apache[ansible_distribution]['package_name']}}"

    - name: Start service and enable start on boot
      service:
        name: "{{apache[ansible_distribution]['service_name']}}"
        state: started
        enabled: yes
```

Listing 6.22 "apache-method-3.yml": Tricky Method of Balancing System Differences

Here, you construct a map and you can access the actual values by using the distribution name as the key. The elegance of this obviously lies in the fact that case distinctions with `when` have become unnecessary.

We could certainly have worked with `ansible_os_family`, which would have spared us the unnecessary distinction between Debian and Ubuntu. Nevertheless, redundancies would still be present because openSUSE in this simple example still has the same settings as the systems from the Debian family.

You could also address this problem with the exotic *Anchors and References* YAML feature from Chapter 4, Section 4.8:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- hosts: all

  vars:
    tmp_anchor1: &package_service_default
      package_name: apache2
      service_name: apache2

    apache:
      Debian: *package_service_default
      Ubuntu: *package_service_default

      Rocky:
        package_name: httpd
        service_name: httpd

      openSUSE Leap: *package_service_default

  tasks:
    - name: Install Apache package
      package:
        name: "{{apache[ansible_distribution]['package_name']}}"

    - name: Start service and enable start on boot
      service:
        name: "{{apache[ansible_distribution]['service_name']}}"
        state: started
        enabled: yes
```

Listing 6.23 "apache-method-3b.yml": Clever Method with Redundancy Avoidance

6.5.4 The Modular Method

The modular method looks like this:

```
---
- hosts: all

  tasks:
    - name: Load system-specific parameters
      include_vars: >
        apache_{{ ansible_distribution | lower | replace(" ", "-") }}.yml

    - name: Install Apache package
      package:
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
        name: "{{apache_package_name}}"

    - name: Start service and enable start on boot
      service:
        name: "{{apache_service_name}}"
        state: started
        enabled: yes
```

Listing 6.24 "apache-method-4.yml": Modular Method of Balancing System Differences

Here, we store the distribution-specific parameters in separate files named *apache_<DISTNAME>.yml*. Since you probably don't want to deal with complicated capitalization and want to deal with spaces in file names even less, I've used some Jinja magic. This is a preview of Section 6.6 and in particular Section 6.6.5.

The files you need to load now are as follows:

- *apache_debian.yml*
- *apache_rocky.yml*
- *apache_opensuse-leap.yml*
- *apache_ubuntu.yml*

The `include_vars` statement searches these files next to the playbook or in a *vars/* subfolder next to the playbook. For better clarity, we usually prefer to use the latter option:

```
---
apache_package_name: apache2
apache_service_name: apache2
```

Listing 6.25 "vars/apache_debian.yml"

```
---
apache_package_name: httpd
apache_service_name: httpd
```

Listing 6.26 "vars/apache_rocky.yml"

```
---
apache_package_name: apache2
apache_service_name: apache2
```

Listing 6.27 "vars/apache_opensuse-leap.yml"

```
---
apache_package_name: apache2
apache_service_name: apache2
```

Listing 6.28 "vars/apache_ubuntu.yml"

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

If you don't want to maintain an additional file for Ubuntu, then of course, you could instead create a symbolic link to the Debian file.

Even with this method, no distinctions are necessary so far. Thus, we have elegance and clarity—which is a good mix!

6.5.5 The Complete Playbook up to This Point

With the new insights, you can now extend the Apache installation to all four systems (including Rocky and SUSE). You have provided the purely Debian/Ubuntu-specific tasks with `when` conditions, and you can now use the generic `package` module for package installation. You'll give the new version the name *apache3.yml*:

```
---
- hosts: all

  tasks:
    - name: Load system-specific parameters
      include_vars: >
        apache_{{ ansible_distribution | lower | replace(" ", "-") }}.yml

    - name: Update package index
      apt:
        update_cache: yes
        cache_valid_time: 3600
      when: ansible_os_family == "Debian"

    - name: Install Apache package
      package:
        name: "{{apache_package_name}}"

    - name: Start service and enable start on boot
      service:
        name: "{{apache_service_name}}"
        state: started
        enabled: yes

    - name: Upload minimal homepage
      copy:
        dest: "{{apache_document_root}}/index.html"
        mode: 0644
        content: |
          <!doctype html>
          <html>
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
            <head>
              <meta charset="utf-8">
              <title>Test</title>
            </head>
            <body>
              <h1>Welcome!</h1>
            </body>
          </html>

    - name: Upload config file
      copy:
        dest: "{{apache_config_directory}}/redirect.conf"
        content: |
          Redirect /go https://www.google.com
      notify: reload apache

    - name: Activate config (only on Debian-like systems)
      command:
        cmd: a2enconf redirect
        creates: /etc/apache2/conf-enabled/redirect.conf
      when: ansible_os_family == "Debian"


  handlers:
    - name: Reload Apache service
      service:
        name: "{{apache_service_name}}"
        state: reloaded
      listen: reload apache
```

Listing 6.29 "apache3.yml": Apache on All Four Distributions

Here, you can see the required vars files:

```
---
apache_package_name: apache2
apache_service_name: apache2
apache_document_root: /var/www/html
apache_config_directory: /etc/apache2/conf-available
```

Listing 6.30 "vars/apache_debian.yml"

```
---
apache_package_name: httpd
apache_service_name: httpd
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
apache_document_root: /var/www/html
apache_config_directory: /etc/httpd/conf.d
```

Listing 6.31 "vars/apache_rocky.yml"

```
---
apache_package_name: apache2
apache_service_name: apache2
apache_document_root: /srv/www/htdocs
apache_config_directory: /etc/apache2/conf.d
```

Listing 6.32 "vars/apache_opensuse-leap.yml"

```
---
apache_package_name: apache2
apache_service_name: apache2
apache_document_root: /var/www/html
apache_config_directory: /etc/apache2/conf-available
```

Listing 6.33 "vars/apache_ubuntu.yml"

6.6 Jinja and Templates

You have already learned about the `copy` module, which allows you to transfer files or content to target hosts. This is certainly a very useful thing, but sometimes, it's not enough to equip all target systems with the same version of a file. At this point, terms like *templates* and *Jinja* come into play, and I would like to explore them in this section.

6.6.1 Definition: Templates and Template Engines

A *template* is a stencil. Advanced Linux users know commands like this:

```
$ find . -name "*.sh" -exec chmod 700 {} \;
```

And they use others like this:

```
$ echo 'Hello %NAME%!' | sed 's/%NAME%/John/'
```

In this context, `chmod 700 {}` and `Hello %NAME%!` are the stencils from which you can obtain a specific result by replacing the placeholders. The `find` and `sed` commands would then be the *template engines*, which are the "machines" that produce a specific, "living" result from the template.

In the world of programming languages, every prominent language also has a template engine. In most cases, this is available as an additional library, and it is not uncommon

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

to have several variants available. In the Python programming language, the *Jinja* template engine (*https://jinja.palletsprojects.com*) is something of a purveyor in terms of templates, and it is therefore also used in Ansible. Since Ansible uses version 2 of this library, it's often referred to as *Jinja2*. We use the terms Jinja and Jinja2 synonymously.

We have encountered Jinja very often, and whenever we implement a variable access in a playbook, we use Jinja:

```
- debug: msg="The color is {{color}}"
```

Now, you might be thinking "Well, is it really so difficult that you need a complete library for simple search and replace?" Certainly not, if that were all. But we are currently using about one percent of Jinja's capabilities, and you will learn a more about the rest in the following pages.

6.6.2 An Individual Homepage for Our Apache Servers

Let's revisit the last state of our Apache example from Section 6.5.5. With the help of Jinja, our Apache servers are about to get a more individualized homepage. We could start right away with the "Set up minimal homepage" task, but let's modify the current task a bit and first outsource the HTML code to an external file:

```
- name: Upload minimal homepage
  template:
    src: index.html.j2
    dest: "{{apache_document_root}}/index.html"
    mode: 0644
```

Now, we're using the `template` module, which "passes" its source file through Jinja. The *.j2* file extension refers to "Jinja2" and indicates that the file is a template file. However, this is not mandatory but just a convention.

The `template` module looks for source files with their relative paths next to the playbook or in a *templates/* subfolder. You should prefer the latter because it's clearer. Place this file there as follows:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>{{inventory_hostname}}: Test</title>
  </head>
  <body>
    <h1>Welcome to {{inventory_hostname}}!</h1>
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
  </body>
</html>
```

Listing 6.34 "templates/index.html.j2": External Jinja Template

Then, run your playbook again and each Apache server should then greet you with its Ansible name. The innovation is currently just a simple variable access, so let's actually use the possibilities of Jinja a bit more, as follows:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>{{inventory_hostname}}: Test</title>
  </head>
  <body bgcolor="{{farbe | default('LightGray')}}">
    <h1>Welcome to {{inventory_hostname}}!</h1>
    <h2>Our team:</h2>
    <ul>
    {% for host in groups['all'] %}
      <li>
      {% if host != inventory_hostname %}
      <a href="http://{{ hostvars[host]['ansible_all_ipv4_addresses'] |
                         map('regex_search','^192\..*')              |
                         select('string')                            |
                         first }}">
        {{host}}
      </a>
      {% else %}
        <span style="background-color: LightCyan">{{host}}</span>
      {% endif %}
      </li>
    {% endfor %}
    </ul>
  </body>
</html>
```

Listing 6.35 "templates/index.html.j2": External Jinja Template with Programming

Here, we use the `if` and `for` – control structures; in essence, Jinja is a programming language. With Ansible, we use *filters* like `default()` a lot, too, and we'll look at this in more detail in the following sections. You can see the results of applying the new version of the template in Figure 6.1.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

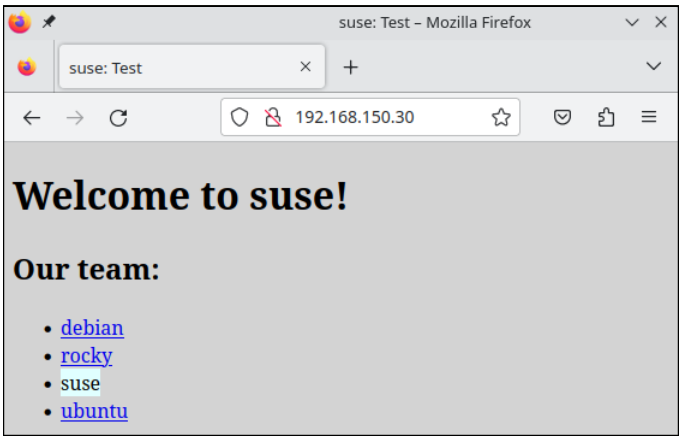


Figure 6.1 Homepage of One of Four Apache Servers

Note

To generate the links, you need the IP address of each host—and this is a more challenging problem than you might think. If you want to test the example outside of our lab environment, filtering by `192` addresses may not be suitable. If you need to, read Chapter 19, Section 19.2.2, as it provides information about other available options.

6.6.3 Quick Testing of Jinja Templates

Sometimes, testing the correct functionality of Jinja templates is not easy. Typically, templating produces files on remote machines; you would have to start some playbook, log into a remote system after its execution, and check whether the desired result was achieved. However, with a few small tricks, you can also test locally and much more directly.

Simple "Inline" Ad Hoc Tests

Jinja supports arithmetic expressions, and this allows you to test whether Jinja is really doing something:

```
$ ansible localhost -m debug -a 'msg={{ 3 + 4 }}'
localhost | SUCCESS => {
    "msg": "7"
}
```

Yes, with this expression, it doesn't matter whether you select `localhost` or any other target host. However, when accessing facts, you need to take more care:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
$ ansible suse -m debug -a 'msg={{ ansible_distribution }}'
suse | SUCCESS => {
    "msg": "openSUSE Leap"
}
```

Important

Accessing facts in an ad hoc command only works if you have a populated and valid facts cache available (see Section 6.3.2)!

Testing Template Files

If your template consists of an entire file (like our Apache homepage), then you need something more sophisticated. Start with this shell script:

```
#!/bin/bash
TARGET_HOST=$1
TEMPLATE_FILE=$2

if [ $# -ne 2 ]; then
    echo "Usage: $0 TARGET_HOST TEMPLATE_FILE" 1>&2
    exit 1
fi

ansible $TARGET_HOST -m debug \
    -a "msg={{lookup('template', '$TEMPLATE_FILE')}}" |
     sed '1s/.*=> //' |
     jq -r '.msg'
```

Listing 6.36 "jinja.sh": Shell Script for Testing Template Files

To use the script, you need the `jq` tool from the package of the same name on the control host; please install it if necessary.

In essence, this is a call to `debug` again, but this time, the script reads a template file by using a lookup, processes it through Jinja, and prints the result. The `sed` command then converts the result into proper JSON, and the JSON `jq` processor selects the content of the `msg` attribute for output.

Now, let's see the whole thing in action. Here is a small template file:

```
Hello from a {{ansible_distribution}}-system!
The sum is: {{3 + 4}}
```

Listing 6.37 "test.j2"

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

And here is the test:

```
$ ./jinja.sh ubuntu test.j2
Hello from a Ubuntu-system!
The sum is: 7
```

Regarding the use of facts, the same limitations mentioned above apply, of course!

6.6.4 Jinja Syntax: Output Expressions, Statements, and Comments

Note

This section is specifically aimed at DevOps with a strong development background!

Essentially, Jinja provides these three syntactic elements:

- The `{{ ... }}` element for expressions that are included in the output after evaluation
- The `{% ... %}` element for statements or control structures
- The `{# ... #}` element for comments that are not included in the output

Expressions

The simplest Jinja expressions are numbers (integers or floating-point numbers), texts (quoted with single or double quotation marks), and the simplest possible Boolean expressions (`true` and `false`).

With these basic building blocks, you can construct more complex expressions. Jinja provides the common options that are available in programming languages:

- **Mathematical/arithmetic expressions**
  You create these with common arithmetic symbols like `+`, `-`, `(...)`, etc.
- **Comparisons**
  You create these with common symbols like `==`, `!=`, etc.
- **Logical combinations**
  You create these with `and`, `or`, and `not`.
- **Various other operators**
  These include `in` (list membership), `|` (filter), and `~` (string concatenation).
- **Lists and maps**
  These include `[ <object1>, <object2>, ...]` and `{ 'key1': <value1>, 'key2': <value2>, ...}`.

The documentation at *https://jinja.palletsprojects.com/templates/#expressions* provides the complete overview.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Instructions

The following structures are available for the flow of control:

? **Assignments with `set`**
Here's an example:

```
{% set number  = 70 %}
{% set animals = ['Dog', 'Cat', 'Mouse'] %}
```

? **Testing conditions with `if`**
Here's an example:

```
{% if number >= 0 and number < 50 %}
The number is too small.
{% elif number >= 50 and number <= 100 %}
The number is large enough.
{% else %}
The number is not in the valid range.
{% endif %}
```

▸ **Loops with `for`**
Example:

```
{% for a in animals %}
* {{ a }}
{% endfor %}
```

? **Subroutines with `macro`**
Here's an example:

```
{% macro greet(x) %}
Hello {{ x }}!
{% endmacro %}

{{ greet('Dog') }}
{{ greet('Cat') }}
```

For more information on macros, Section 6.6.7.

You can find a complete overview of all control structures at *https://jinja.palletsprojects.com/templates/#list-of-control-structures*.

6.6.5 Filter

When you're working with Ansible, you'll find that Jinja filters are often useful; in syntax and semantics, they are comparable to the UNIX pipe mechanism. Here are some

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

examples, starting with the `int` filter, which converts floating-point numbers into integers:

```
{{ (10 / 3) | int }}    {# --> 3 #}
```

The `join()` filter "glues" list elements into a text using a provided separator:

```
{% set animals = ['Dog', 'Cat', 'Mouse'] %}

{{ animals | join(', ') }}    {# --> "Dog, Cat, Mouse" #}
```

The `upper` and `lower` filters convert text to uppercase or lowercase:

```
{{ "Some Text" | upper }}    {# --> "SOME TEXT" #}
{{ "Some Text" | lower }}    {# --> "some text" #}
```

The `default()` filter provides a value in case of an undefined variable:

```
{{ a_number | default(42) }}
```

And the `replace()` filter implements a search-and-replace function:

```
{{ "This is wrong" | replace("wrong": "correct") }}
```

There are about fifty of these filters in Jinja, and of course, they have their own section in the official Jinja documentation: *https://jinja.palletsprojects.com/templates/#builtin-filters*. Please take the opportunity to get a general overview there.

Ansible adds many more filters. There are filters for generating (password) hashes like `hash()` and `password_hash()`, filters for path name analysis like `basename` and `dirname`, filters for URL analysis like `urlsplit`, filters for regular expressions like `regex_search()` and `regex_replace()`, and many more.

Quite often, you will encounter the `bool` filter, which converts textual representations of truth values such as `"True"`: `"FALSE"`: `"yes"`: and `"0"` into actual `boolean` values:

```
{% if "false" | bool  %}Correct!{% else %}Wrong!{% endif %}
```

(Try it without the `bool` filter and you will see the difference.)

And again, I would like to refer you to the official online documentation: *https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html*.

6.6.6 Whitespace Control

Sometimes, you need to be very meticulous to ensure that your Jinja template does not produce unwanted whitespaces. Even a newline at the end of a filename will lead to annoying and difficult-to-diagnose problems.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

However, with Jinja, you can influence the handling of whitespaces. Here is a small playbook that you can use to experiment with whitespace control:

```
---
- hosts: localhost
  gather_facts: false

  tasks:
    - debug:
        msg: |-
          {# Just a comment #}

          Hello
          {% if True %}
              Jinja
          {% else %}
              This will never appear
          {% endif %}
          !
```

Listing 6.38 "whitespace.yml": Jinja Template That Produces Unwanted Whitespaces

As a result, a whole lot of whitespaces end up in the output:

```
$ ansible-playbook whitespace.yml
ok: [localhost] => {
    "msg": "\nHello\n    Jinja\n!"
}
```

If you need more control, you can equip all Jinja separators (*delimiters*) with an added minus sign, as follows:

- {{- or -}}
- {%- or -%}
- {#- or -#}

This will remove all whitespaces in the respective "direction" (i.e., spaces, tabs, and newlines) from the output. You'll get a clean single-line output by using this:

```
---
- hosts: localhost
  gather_facts: false

  tasks:
    - debug:
        msg: |-
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
{# Just a comment -#}

Hello {{ "" }}
{%- if True -%}
    Jinja
{%- else %}
    This will never appear
{% endif -%}
!
```

Listing 6.39 "whitespace-2.yml": Suppression of Unwanted Whitespaces

```
$ ansible-playbook whitespace-2.yml
ok: [localhost] => {
    "msg": "Hello Jinja!"
}
```

6.6.7 Macros

To wrap up our discussion of Jinja, I'd like to introduce you to a powerful technique that can help bring more clarity and reduce redundancies in larger templating projects. We are talking about *macros*, which are similar to functions or subroutines you'll know from other programming languages.

Jinja macros are (unfortunately) not global; instead, they exist only where they are instantiated. To actually avoid redundancies, you typically outsource into separate files, like this:

```
{% macro frame(text) -%}
+{{ "-" * (text | length + 2) }}+
| {{text}} |
+{{ "-" * (text | length + 2) }}+
{% endmacro %}


{% macro add(a, b) -%}
{{ a + b }}
{%- endmacro %}
```

Listing 6.40 "utils.j2": External Macro Collection

And when they are needed, you import them first:

```
---
- hosts: localhost
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
  tasks:
    - debug:
        msg: |-
          {% import 'utils.j2' as utils %}
          {{ utils.add(3, 4) }}

    - copy:
        dest: /tmp/frame.txt
        content: |-
          {% import 'utils.j2' as utils %}
          {{ utils.frame("Hello macro!") }}
```

Listing 6.41 "macro-test.yml": Integrating External Macro Collection

The result looks like this:

```
$ ansible-playbook macro-test.yml && cat /tmp/frame.txt
[…]
ok: [localhost] => {
    "msg": "7"
}
[…]

+--------------+
| Hello macro! |
+--------------+
```

6.7 Variable Tests

Now that you've learned a lot about variables, let's take a very detailed look at the possible states of a variable. A variable in a play can have any of the following characteristics:

- It can not exist at all.
- It can exist but have no value.
- It can exist and have a value.

In larger projects, you must be able to distinguish between these possibilities precisely. In the playbook from Listing 6.42, I provide you with a summary of the techniques you need for this. All `debug` tasks output their message—with the exception of the two tasks that "rescue" themselves with `ignore_errors`. Otherwise, these would terminate the play with a runtime error.

Using the `NULL` literal is one of many ways that you can describe a nonexistent value in YAML. See Chapter 4, Section 4.7.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts: localhost
  gather_facts: no

  vars:
    # var1 has no value:
    var1: NULL

    # var2 is not even there.

  tasks:
    # 'debug: var=' never has a problem:
    - name: debug var var1
      debug: var=var1

    - name: debug var var2
      debug: var=var2

    # Direct Jinja-access is only possible to known variables:
    - name: debug msg var1
      debug: msg={{ var1 }}

    - name: debug msg var2
      debug: msg={{ var2 }}
      ignore_errors: true

    # Ansible when-checks:
    - name: when check (1)
      debug: msg="var1 is defined, but has no value"
      when: var1 is defined and var1 == None

    - name: when check (2)
      debug: msg="var2 is not defined"
      when: var2 is undefined

    - name: when check (3)
      debug: msg="Comparing undefined variables with None is not allowed"
      when: var2 == None
      ignore_errors: true

    # Same with Jinja conditionals:
    - name: Jinja if check (1)
      debug:
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
      msg: |-
        {% if var1 is defined and var1 == None %}
        Jinja: var1 is defined, but has no value
        {%- endif %}
  - name: Jinja if check (2)
    debug:
      msg: |-
        {% if var2 is undefined %}
        Jinja: var2 is not defined
        {%- endif %}

  # Assign value to var2:
  - set_fact:
      var2: 'Hello'

  - debug: msg="var2 is now defined and has a value"
    when: var2 is defined and var2 != None
```

Listing 6.42 "null.yml": Various Variable Tests

Important

In Ansible, there is *no* way to make a variable disappear once it has been created (there is nothing like `unset_fact`).

In the above playbook, you can never go back to the `var2 is undefined` state in subsequent tasks, no matter what you try.

6.8 Lookup Plug-ins

Lookup plug-ins (or simply *lookups*) are an Ansible-specific extension of Jinja2 (Section 6.6). With them, you can access a wide variety of external data sources, but this is always *done by the control host*. The results of the lookups are typically written into variables or directly processed in templates.

Look at the first example in Listing 6.43, in which you're using a `pipe` lookup to access the output of an external command:

```
---
- hosts: localhost

  vars:
    datetime: "{{ lookup('pipe', 'date') }}"
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
  tasks:
    - debug: var=datetime
    - pause: seconds=2
    - debug: var=datetime
```

Listing 6.43 "lookup_pipe.yml": "pipe" Lookup

The target host is not bothered in this example; nothing in this play happens outside of the control host. Maybe it's a bit unexpected that the lookup is executed again and again each time the variable is accessed:

```
$ ansible-playbook lookup_pipe.yml
TASK [debug] *******************************************************************
ok: [localhost] => {
    "datetime": "Mon Oct 14 07:51:17 PM UTC 2024"
}

TASK [pause] *******************************************************************
Pausing for 2 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [localhost] => {
    "datetime": "Mon Oct 14 07:51:19 PM UTC 2024"
}
```

Unfortunately, the complete list of all lookups is no longer available online; instead, you must refer to the offline documentation:

```
$ ansible-doc -t lookup -l
```

In Listing 6.44, you'll see some more examples of lookups:

```
---
- hosts: localhost

  tasks:
    - name: Environment lookup
      debug:
        msg: "{{ lookup('env', 'HOME') }}"

    - name: File lookup
      debug:
        msg: "{{ lookup('file', '/etc/os-release') }}"

    - name: Filename globbing
      debug:
        msg: "{{ lookup('fileglob', '/etc/*.conf') }}"
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
      # For a 'true list' pass wantlist=True to the lookup.

  - name: List /usr/local recursively (may result in many files)
    debug:
      msg: "{{ lookup('community.general.filetree', '/usr/local') }}"

  - name: CSV lookup
    debug:
      msg: "{{ lookup('csvfile',
                      'root file=/etc/passwd delimiter=: col=6') }}"

  - name: DNS lookup
    debug:
      msg: "{{ lookup('dig', 'www.example.com') }}"
```

Listing 6.44 "lookups.yml": Various Lookups

6.9 Loops

Ansible's ability to repeat individual tasks is just as useful as testing conditions. In developer jargon, this is called a *loop* or (if you like it to sound a bit more scientific) *iteration*. Among other things, you can iterate over lists or maps. Examples of this include creating many users with the `user` module or changing the permissions of many files with the `file` module. You can also have tasks repeat until a certain condition is met. I will now provide examples for these concepts, and you can find a complete summary at *https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html*.

I'll start by mentioning that there are two types of loops in Ansible:

- This one has always been available:

  `with_*`

- And this one has been available since Ansible 2.5:

  `loop`

Ansible recommends the newer `loop` variant in most cases. However, there are currently no plans to discontinue support for the classic variant, and since I personally don't see any real advantages in the new syntax from a practical perspective, I will focus on the old form here. If in doubt, refer to the official documentation mentioned above to form your own opinion.

6.9.1 "with_*" Loops Are Lookups

Technically, all classic `with_*` loops are implemented as lookups (Section 6.8). In theory, you can use any lookup as a loop, but in practice, there are only a few where both types

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

of usage are equally plausible. An example of this is the `fileglob` lookup, as the following listing shows:

```
- hosts: localhost

  tasks:
    - name: Store list of .conf filenames in variable
      set_fact:
        configs: "{{ lookup('fileglob', '/etc/*.conf', wantlist=True) }}"

    - name: Create filelist
      copy:
        dest: /tmp/filelist.txt
        content: |
          Found the following files:
          ==========================
          {% for file in configs %}
          - {{ file }}
          {% endfor %}


    - name: Delete all .bak and .old files from home directory
      file:
        path: "{{item}}"
        state: absent
      with_fileglob:
        - ~/*.bak
        - ~/*.old
```

Listing 6.45 "fileglob.yml", Lookups and Loops—Two Sides of Same Coin

6.9.2 Iteration Over a List with "with_items" or "with_list"

With the `with_items` loop type, you can iterate over a list as follows:

```
---
- hosts: localhost
  vars:
    list:
      - Dog
      - Cat
      - Mouse

  tasks:
    - name: List loop
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
      debug: msg="Hello {{item}}!"
      with_items: "{{list}}"
```

Listing 6.46 "list-loop.yml": Example of "with_items" Loop

The `item` variable is implicitly the "magic" loop variable. It contains the current list value in each loop iteration, and this is how it looks:

```
$ ansible-playbook list-loop.yml
[…]
TASK [List loop] ***********************************************************
ok: [localhost] => (item=Dog) => {
    "msg": "Hello Dog!"
}
ok: [localhost] => (item=Cat) => {
    "msg": "Hello Cat!"
}
ok: [localhost] => (item=Mouse) => {
    "msg": "Hello Mouse!"
}
[…]
```

But let's investigate this further. How does `with_items` behave when the individual list elements are lists themselves? In the following example, would you expect two loop iterations or five?

```
---
- hosts: localhost
  vars:
    list:
      - [Dog, Cat, Mouse]
      - [Ant, Donkey]

  tasks:
    - name: Nested list loop
      debug: msg="Hello {{item}}!"
      with_items: "{{list}}"
```

Listing 6.47 "flatten-list-loop.yml": "with_items" Loop of Nested Lists

If you try it, you will find that there are five loop iterations:

```
$ ansible-playbook flatten-list-loop.yml
[…]
TASK [Nested list loop] ****************************************************
ok: [localhost] => (item=Dog) => {
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    "msg": "Hello Dog!"
}
ok: [localhost] => (item=Cat) => {
    "msg": "Hello Cat!"
}
ok: [localhost] => (item=Mouse) => {
    "msg": "Hello Mouse!"
}
ok: [localhost] => (item=Ant) => {
    "msg": "Hello Ant!"
}
ok: [localhost] => (item=Donkey) => {
    "msg": "Hello Donkey!"
}
[…]
```

This is no bug but a feature called *list flattening*. If you don't want this, then you can use the `with_list` loop, which is closely related and which works without flattening:

```
---
- hosts: localhost
  vars:
    list:
      - [Dog, Cat, Mouse]
      - [Ant, Donkey]

  tasks:
    - name: List loop without flattening
      debug: msg="Hello {{item}}!"
      with_list: "{{list}}"
```

Listing 6.48 "noflatten-list-loop.yml": Example of "with_list" Loop

The result looks like this:

```
$ ansible-playbook noflatten-list-loop.yml
[…]
TASK [List loop without flattening] ****************************************
ok: [localhost] => (item=['Dog', 'Cat', 'Mouse']) => {
    "msg": "Hello ['Dog', 'Cat', 'Mouse']!"
}
ok: [localhost] => (item=['Ant', 'Donkey']) => {
    "msg": "Hello ['Ant', 'Donkey']!"
}
[…]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

6.9.3 Iteration Over a Map with "with_dict"

Here, you see an example that iterates over a map with `with_dict`:

```
---
- hosts: localhost
  vars:
    map:
      Dog: woof
      Cat: meow
      Mouse: squeak

  tasks:
    - name: Map loop
      debug: msg="{{ item.key }} says {{ item.value }}."
      with_dict: "{{map}}"
```

Listing 6.49 "map-loop.yml": Example of "with_dict" Loop

Here, too, there is the magical `item` variable, and this time, it consists of a key-value pair. You can then access the part you want with the `key` and `value` selectors:

```
$ ansible-playbook map-loop.yml
[…]
TASK [Map loop] ************************************************************
ok: [localhost] => (item={'key': 'Dog', 'value': 'woof'}) => {
    "msg": "Dog says woof."
}
ok: [localhost] => (item={'key': 'Cat', 'value': 'meow'}) => {
    "msg": "Cat says meow."
}
ok: [localhost] => (item={'key': 'Mouse', 'value': 'squeak'}) => {
    "msg": "Mouse says squeak."
}
[…]
```

6.9.4 Iteration Over a Generated Sequence with "with_sequence"

With the `with_sequence` loop type, you can generate ranges and iterate over them. Here's an example:

```
---
- hosts: localhost

  tasks:
    - name: Sequence loop
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    debug: msg="{{item}}"
    with_sequence: start=6 end=12 stride=2 format=test%02d.txt
```

Listing 6.50 "sequence-loop.yml": Example of "with_sequence" Loop

All parameters have sensible defaults. Instead of `end`, you can specify `count` to indicate how many iterations you need. You can specify formats in the usual `printf` syntax; without a special format, the result is just numbers. See the output of the above example:

```
$ ansible-playbook sequence-loop.yml
[…]
TASK [Sequence loop] ***********************************************************
ok: [localhost] => (item=test06.txt) => {
    "msg": "test06.txt"
}
ok: [localhost] => (item=test08.txt) => {
    "msg": "test08.txt"
}
ok: [localhost] => (item=test10.txt) => {
    "msg": "test10.txt"
}
ok: [localhost] => (item=test12.txt) => {
    "msg": "test12.txt"
}
[…]
```

6.9.5 Looping Over the Combination of Two Lists with "with_nested"

```
---
- hosts: localhost

  tasks:
    - name: Loop over combination of lists
      debug: msg="{{ item[0] }} likes {{ item[1] }}"
      with_nested:
        - [Dog, Cat, Mouse]
        - [Chips, Popcorn]
```

Listing 6.51 "nested-loop.yml": Example of "with_nested" Loop

```
$ ansible-playbook nested-loop.yml
[…]
TASK [Loop over combination of lists] ******************************************
ok: [localhost] => (item=['Dog', 'Chips']) => {
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    "msg": "Dog likes Chips"
}
ok: [localhost] => (item=['Dog', 'Popcorn']) => {
    "msg": "Dog likes Popcorn"
}
ok: [localhost] => (item=['Cat', 'Chips']) => {
    "msg": "Cat likes Chips"
}
[…]
```

6.9.6 Looping Over Two Parallel Lists with "with_together"

```
---
- hosts: localhost

  tasks:
    - name: Loop over parallel lists
      debug: msg="{{ item[0] }} is number {{ item[1] }}"
      with_together:
        - [Dog, Cat, Mouse]
        - [one, two, three]
```

Listing 6.52 "together-loop.yml": Example of "with_together" Loop

```
$ ansible-playbook together-loop.yml
[…]
TASK [Loop over parallel lists] ********************************************
ok: [localhost] => (item=['Dog', 'one']) => {
    "msg": "Dog is number one"
}
ok: [localhost] => (item=['Cat', 'two']) => {
    "msg": "Cat is number two"
}
ok: [localhost] => (item=['Mouse', 'three']) => {
    "msg": "Mouse is number three"
}
[…]
```

6.9.7 Nested Loops with "with_subelements"

The use of the `with_subelements` loop type is more sophisticated in some ways: you have a list of objects (which are maps) that have a specific attribute, which is itself a list. (These could be users with various SSH keys, Apache virtual hosts with different aliases, and much more.) For simplicity's sake, we use dogs, cats, and mice again:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts: localhost
  vars:
    data:
      - name: Dog
        sounds:
          - woof
          - growl
      - name: Cat
        sounds:
          - meow
      - name: Mouse

  tasks:
    - name: Loop over subelements
      debug: msg="{{ item[0].name }} says {{ item[1] }}."
      with_subelements:
        - '{{data}}'
        - sounds
        - { skip_missing: 1 }
```

Listing 6.53 "subelements-loop.yml": Example of "with_subelements" Loop

The third entry under `with_subelements` is optional. If it were missing, the playbook would fail because there are no sounds for the mouse. However, it runs without error and the mouse does not appear at all:

```
$ ansible-playbook subelements-loop.yml
[…]
TASK [Loop over subelements] ***************************************************
ok: [localhost] => (item=[{'name': 'Dog'}, 'woof']) => {
    "msg": "Dog says woof."
}
ok: [localhost] => (item=[{'name': 'Dog'}, 'growl']) => {
    "msg": "Dog says growl."
}
ok: [localhost] => (item=[{'name': 'Cat'}, 'meow']) => {
    "msg": "Cat says meow."
}
[…]
```

6.9.8 Repeating Tasks with "until"

Here is an example of the `until` loop type:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- hosts: localhost

  tasks:
    - name: Roll the dice
      shell: "echo $(( $RANDOM % 6 + 1))"
      register: result
      args:
        executable: /bin/bash
      until: "'6' in result.stdout"
      retries: 10
      delay: 1

    - debug: msg="A six was rolled!"
```

Listing 6.54 "'until-loop.yml": Example of "until" loop

With this loop type, you can repeat a task until a certain condition is met. However, having infinite loops (i.e., waiting forever for a condition that never occurs) is not possible in Ansible. This is ensured by the `retries` parameter, with which you must set a maximum number of attempts. You also set the interval between two attempts (in seconds) with `delay`. Of course, there are default values for these two parameters:

- Default value for `retries`: 3
- Default value for `delay`: 5

If no success occurs after `retries` attempts, the task ends with an error. In that case, however, the caller was lucky:

```
$ ansible-playbook until-loop.yml
[…]
TASK [Roll the dice] ***********************************************************
FAILED - RETRYING: [localhost]: Roll the dice (10 retries left).
FAILED - RETRYING: [localhost]: Roll the dice (9 retries left).
FAILED - RETRYING: [localhost]: Roll the dice (8 retries left).
FAILED - RETRYING: [localhost]: Roll the dice (7 retries left).
changed: [localhost]

TASK [debug] *******************************************************************
ok: [localhost] => {
    "msg": "A six was rolled!"
}
[…]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

6.9.9 More Control with "loop_control"

With the `loop_control` task parameter, you can further control the behavior of some loop types. I will now show you some options.

Renaming the Loop Variable with "loop_var"

Sometimes, you may want to rename the magic `item` loop variable—and in rare cases, you *have to* do it (for example, in a loop over `include_tasks` statements where the included tasks also use loops internally). Renaming is quite simple with `loop_control` and the `loop_var` subdirective:

```
---
- hosts: localhost
  vars:
    list:
      - Dog
      - Cat
      - Mouse

  tasks:
    - name: List loop
      debug: msg="Hello {{a}}!"
      with_items: "{{list}}"
      loop_control:
        loop_var: a
```

Listing 6.55 "loop_var.yml": Renaming Loop Variable

Output Limitation with "label"

When you're looping over larger objects, Ansible often generates unpleasant amounts of output, as in this case:

```
---
- hosts: debian

  vars:
    users:
      - name: john
        home: /home/john
        pass: secret
      - name: mary
        shell: /bin/sh
        pass: supersecret
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
  tasks:
    - name: Create user
      user:
        name:     '{{item.name}}'
        shell:    '{{item.shell | default("/bin/bash")}}'
        home:     '{{item.home  | default(omit)}}'
        password: '{{item.pass  | password_hash("sha512") }}'
        update_password: on_create
      with_items: '{{users}}'
```

Listing 6.56 "create-users.yml"

```
$ ansible-playbook create-users.yml
[…]
TASK [Create user] *********************************************************
changed: [debian] => (item={'name': 'john', 'home': '/home/john',
                            'pass': 'secret'})
changed: [debian] => (item={'name': 'mary', 'shell': '/bin/sh',
                            'pass': 'supersecret'})

[…]
```

Even worse, not only too much data but even sensitive data is logged. A `loop_control/label` provides a remedy because it allows you to choose the attribute to be logged:

```
# [...]
      with_items: '{{users}}'
      loop_control:
        label: '{{item.name}}'
```

This results in a less extensive output:

```
$ ansible-playbook create-users.yml
[…]
TASK [Create user] *********************************************************
changed: [debian] => (item=john)
changed: [debian] => (item=mary)
[…]
```

Note

Occasionally, you'll get a recommendation to use `no_log: true` to prevent the output of sensitive data. You can do that—but then, nothing will be logged anymore. For example, if the properties of a user were to change, you would see the following:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
TASK [Create user] *********************************************
ok: [debian] => (item=None)
changed: [debian] => (item=None)
```

You can probably guess which user caused the change now....

Tracking the Index Position with "index_var"

With `loop_control/index_var`, you can instruct Ansible to provide the position of the currently processed element during iteration:

```
---
- hosts: localhost
  vars:
    list:
      - Dog
      - Cat
      - Mouse

  tasks:
    - name: List loop
      debug: msg="Animal No. {{num + 1}} is {{ item }}."
      with_items: "{{list}}"
      loop_control:
        index_var: num
```

Listing 6.57 "index_var.yml": Loop Counter with "loop_control.index_var"

Ansible starts counting at 0, so I did a little cleanup of the output. It looks like this:

```
$ ansible-playbook index_var.yml
[…]
TASK [List loop] ***********************************************************
ok: [localhost] => (item=Dog) => {
    "msg": "Animal No. 1 is Dog."
}
ok: [localhost] => (item=Cat) => {
    "msg": "Animal No. 2 is Cat."
}
ok: [localhost] => (item=Mouse) => {
    "msg": "Animal No. 3 is Mouse."
}
[…]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

6.9.10 "register" plus Loop

You might wonder whether it's feasible to attach a loop to a task and store the results in a variable using `register`. A quick test reveals that Ansible does, in fact, have a solution for this scenario:

```
---
- hosts: debian
  gather_facts: no

  tasks:
    - command: du -sh {{item}}
      with_items:
        - /etc
        - /home
        - /var
      changed_when: false
      register: du_cmd

    - debug: var=du_cmd
```

Listing 6.58 "register_loop.yml": Example for Combination of "register" and Loop

You'll see output that's similar to this:

```
[…]
ok: [debian] => {
    "du_cmd": {
        "changed": false,
        "msg": "All items completed",
        "results": [
            {
                "ansible_loop_var": "item",
                "item": "/etc",
                "changed": false,
                "cmd": [
                    "du",
                    "-sh",
                    "/etc"
                ],
                [...]
            },

    {
                "ansible_loop_var": "item",
                "item": "/home",
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
                [...]
            },
[…]
```

The `du_cmd` result contains a `results` list whose elements are in turn maps that correspond to the individual loop iterations. I recommend that you take a close look at such a `register` `results` list in case you need to use it in future. It will then be easy for you to select the parts you want!

6.10 Error Handling with "failed_when" and "ignore_errors"

If you ever want to define for yourself what qualifies as an error in a task, you can use the `failed_when` directive. Especially in `command` or `shell` tasks, this is particulary useful when the executed commands have a different idea of what is considered faulty than you.

For example, it is technically not an error for the `find` command to find nothing. We change this (from Ansible's perspective) in this example:

```
---
- hosts: debian

  tasks:
    - name: Let find fail when nothing is found
      command: find /etc -name foo.conf
      changed_when: false
      register: find_cmd
      failed_when: find_cmd.stdout == '' or find_cmd.rc != 0

    - debug: msg={{ find_cmd.stdout }}
```

Listing 6.59 "failed_when.yml": Defining Task's Error Status with "failed_when"

Unless there's actually a file named *foo.conf* in some */etc* directory, the play will stop after the first task.

Ansible offers another task parameter called `ignore_errors`, but it's used only rarely. If you set this parameter to `yes`/`true`, Ansible will ignore any errors in this task. The risk of doing this is that the play will continue, even though a potentially serious problem may have occurred. Here is a very simple example:

```
---
- hosts: localhost
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
  tasks:
    - name: This task would normally fail
      command: /bin/false
      changed_when: false
      ignore_errors: true
```

Listing 6.60 "ignore_errors.yml": Ignoring Errors with "ignore_errors"

This is what it looks like:

```
$ ansible-playbook ignore_errors.yml
[…]
TASK [This task would normally fail] ************************************
fatal: [localhost]: FAILED! => {"changed": false, "cmd": ["/bin/false"], ...
...ignoring

PLAY RECAP ********************************************************************
localhost                  : ok=2  [...]          failed=0  [...]  ignored=1
```

6.11 Blocks

One of the many new features in Ansible 2.0 was the ability to group tasks into logical blocks. Here is a simple example of using the `block` directive that should demonstrate the directive's usefulness:

```
---
- hosts: all

  tasks:
    - name: Debian-related Tasks
      block:
      - name: Only on Debian-like systems (1)
        debug: msg="Hello"

      - name: Only on Debian-like systems (2)
        debug: msg="Hello again"

      when: ansible_os_family == "Debian"
```

Listing 6.61 "block.yml": Summary of Tasks in Block

Now, you can specify parameters like `when`, `ignore_errors`, and `become_user` at the block level, instead of having to repeat them in every single task. That was certainly a feature the world was waiting for back then.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Since version 2.3, blocks, like regular tasks, can also have a `name` field. The only purpose of this field is documentation. and such a block name has no other visible external effect. (It's not displayed during the execution of the playbook.) Therefore, it's often omitted. The idea of also displaying the block name was discussed years ago, but it was ultimately rejected due to technical reasons.

Blocks also have useful properties for error handling (the keyword is *exception handling*). In a `block`, you can define an optional `rescue` section that is roughly equivalent to a `catch` in many common programming languages. If any of the `block` tasks encounter an error, Ansible will branch into this `rescue` section, and for the corresponding hosts, normal processing will continue there.

You can do anything in a `rescue` section; typically, you'll try to somehow respond to the problem or simply clean up or maybe "roll back," as the following simple example demonstrates:

```
- hosts: debian

  tasks:
    - block:
      - name: Create directory
        file:
          path: /tmp/download
          state: directory

      - name: Download some files from another server
        get_url:
          url: http://192.168.150.20/{{item}}
          dest: /tmp/download/{{item}}
        with_items: [index.html, test42.txt]

      rescue:
      - debug: msg="Something went wrong"

      - name: Clean up
        file:
          path: /tmp/download
          state: absent
```

Listing 6.62 "rescue.yml": Block with "rescue" Section

Here are two additional notes before we close this section:

- In addition to `rescue`, Ansible offers an `always` section, in which instructions are *always* executed, regardless of whether a `block` instruction has failed or not.
- An unreachable host does not trigger a rescue jump!

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

6.12 Timeouts and Asynchronous Execution

If you're dealing with very time-consuming tasks, you can limit their time or even run them asynchronously in the background and let the `timeout`, `async`, and `poll` parameters control them.

Let me first note that in principle, an Ansible task can run "forever" if it is not limited internally by specific mechanisms or if the SSH connection is not eventually terminated due to "inactivity." Ever since Ansible 2.10, the `timeout` directive has been available at the task level, as the following example demonstrates:

```
---
- hosts: localhost

  tasks:
    - name: Do nothing for a day ...
      command: sleep 86400
      timeout: 10
      ignore_errors: true

    - debug: msg="Let's continue ..."
```

Listing 6.63 "timeout.yml": Cancelling Long-Running Tasks with "timeout"

If the maximum runtime specified in seconds is exceeded, the task will fail.

You can achieve similar effects with the `async` and `poll` directives, and you can even run tasks asynchronously (i.e., in the background). When you're using `async` , you specify a *maximum runtime* and the `poll` parameter determines *the interval at which the status should be checked internally*. (The default is every ten seconds.) Table 6.3 shows which scenarios are possible here.

Value for async	Value for poll	Effects
Not present or equal to 0	Doesn't matter	There is normal (synchronous) execution.
Greater than 0	Greater than 0	After a maximum of `async` seconds, the task will be terminated. Every `poll` seconds, a check is performed (this keeps the connection open to prevent potential SSH timeouts from occurring). It feels like a normal synchronous execution or a timeout.
Greater than 0	0	Start the task and go straight to the next one. The task will also be terminated here after `async` seconds if necessary.

Table 6.3 Possible Combinations of "async" and "poll"

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

If necessary, you can also wait for the completion of asynchronous tasks later (which is, in a way, the real art). In Listing 6.64, you'll see a complete example with two tasks:

```
---
- hosts: localhost

  tasks:
    - name: This task takes about 10 seconds
      shell: for i in $(seq 10); do date >>/tmp/task1; sleep 1; done
      args:
        executable: /bin/bash
      async: 1000
      poll: 0
      register: task1

    - name: This task takes about 15 seconds
      shell: for i in $(seq 15); do date >>/tmp/task2; sleep 1; done
      args:
        executable: /bin/bash
      async: 1000
      poll: 0
      register: task2

    - name: Check task 1
      async_status: jid={{ task1.ansible_job_id }}
      register: result1
      until: result1.finished
      retries: 30
      delay: 1

    - name: Check task 2
      async_status: jid={{ task2.ansible_job_id }}
      register: result2
      until: result2.finished
      retries: 30
      delay: 1
```

Listing 6.64 "async.yml": Example of Asynchronous Tasks: Two Asynchronous Tasks

You'll see two longer-running tasks that are simulated by small shell one-liners. In total, the two tasks would take about 25 seconds to complete with normal processing, but due to asynchronous processing, the play is completed in just over 15 seconds.

You can compact waiting for the completion of individual jobs by combining `with_items` and `until`:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- hosts: localhost

  tasks:
    - name: This task takes about 10 seconds
      shell: for i in $(seq 10); do date >>/tmp/task1; sleep 1; done
      args:
        executable: /bin/bash
      async: 1000
      poll: 0
      register: task1

    - name: This task takes about 15 seconds
      shell: for i in $(seq 15); do date >>/tmp/task2; sleep 1; done
      args:
        executable: /bin/bash
      async: 1000
      poll: 0
      register: task2

    - name: Check tasks
      async_status: jid={{ item.ansible_job_id }}
      register: result
      until: result.finished
      retries: 30
      delay: 1
      with_items:
        - "{{task1}}"
        - "{{task2}}"
```

Listing 6.65 "async-v2.yml": Example of Asynchronous Tasks with Optimized Status Check

You'll see this in practice only rarely. For example, you could conceivably start a very time-consuming download and do something else while it runs—provided that you have something to do and don't want to just wait for the download.

6.13 Local Tasks

So far, our control host has mainly acted as a conductor with the primary role of telling the target systems what to do—but sometimes, it also has to lend a helping hand. For example, the control host could first download a file from the internet and then distribute it to the target hosts (because they might not have direct internet access or it wouldn't be efficient for 100 servers to pull the same resource from the internet).

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

You can quickly construct a first solution; you can distribute the tasks well by dividing them into two plays:

```
---
- hosts: localhost

  tasks:
    - get_url:
        url: https://upload.wikimedia.org/wikipedia/commons/a/af/Tux.png
        dest: /tmp/tux.png


- hosts: all:!suse

  tasks:
    - copy:
        src: /tmp/tux.png
        dest: /var/www/html
        mode: 0644
```

Listing 6.66 "controlhost-action.yml": Involving the Control Host

We have excluded the SUSE machine here because the target directory would have to be a different one. This is certainly not a huge problem, but it would have complicated the example unnecessarily.

It's not always the best choice to cut a play in two. Perhaps you're dealing with `hosts: all:!suse` and it would be very unpleasant to interrupt the play because of this small action. One solution is to use the `delegate_to` Ansible directive:

```
- hosts: all:!suse

  tasks:
    - name: Download penguin image
      delegate_to: localhost
      get_url:
        url: https://upload.wikimedia.org/wikipedia/commons/a/af/Tux.png
        dest: /tmp/tux.png
      run_once: true

    - name: Upload to target hosts
      copy:
        src: /tmp/tux.png
        dest: /var/www/html
        mode: 0644
```

Listing 6.67 "delegate_to-localhost.yml": Involving the Control Host with "delegate_to"

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The effect of `delegate_to` is to assign the task to another target host (in this case: `localhost`). This one apparently doesn't even have to be on the host list of the current play.

You'll also see the new `run_once` task parameter, which prevents our control host from doing the same thing three times because the first task would otherwise be executed normally for *all* target systems.

6.13.1 "local_action"

The `local_action` directive is available as an alternative to `delegate_to: localhost`. *This variant is no longer recommended*, but I don't want to withhold it from you entirely, as it is still often encountered:

```
- hosts: all:!suse

  tasks:
    - name: Download penguin image
      local_action:
        module: get_url
        url: https://upload.wikimedia.org/wikipedia/commons/a/af/Tux.png
        dest: /tmp/tux.png
      run_once: true

    - name: Upload to target hosts
      copy:
        src: /tmp/tux.png
        dest: /var/www/html
        mode: 0644
```

Listing 6.68 "local_action.yml": Involving Control Host with "local_action"

6.13.2 Using Other Hosts

Could the `ubuntu` host, for example, download the image and distribute it to its "colleagues?" Yes and no. I would advise you to take a small detour via the control host here:

1. `ubuntu` downloads the image.
2. `localhost` retrieves the image from `ubuntu` with *fetch*.
3. `localhost` distributes the image as usual with *copy*.

In theory, you could use more direct solutions with `rsync` or the `synchronize` module, but they always require a password-free SSH key relationship *between the target hosts*, which is by no means always given.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

6.14 Environment Variables

When you deal with the topic of *environment variables*, it usually involves one of the following two concerns:

- You want to set an environment variable for a task (or a series of tasks).
- You want to read an environment variable and process its value.

6.14.1 Setting Environment Variables with the "environment" Parameter

With the `environment` parameter, you can set environment variables for target systems during the runtime of a playbook, at the level of individual tasks, blocks, or an entire play.

Here's an example of setting an environment variable at the task level:

```
---
- hosts: debian

  tasks:
    - shell: echo $ANIMAL
      changed_when: false
      register: echo_cmd
      environment:
        ANIMAL: Dog

    - debug: var=echo_cmd.stdout
```

Listing 6.69 "env_task.yml": Setting Environment Variable at Task Level

This works similarly at the block or play level. A typical use case is setting a proxy that allows target systems to access the internet. This could look something like this:

```
---
- hosts: all

  environment:
    http_proxy:  http://proxy.example.com:3128
    https_proxy: http://proxy.example.com:3128

  tasks:
    [...]
```

Listing 6.70 "some_playbook.yml": Example Playbook Excerpt

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

You could then define the map with the required environment variables somewhere *outside* to reduce redundancy (e.g., in *group_vars/**). The playbook would then look something like this:

```
---
- hosts: all

  environment: "{{my_env}}"

  tasks:
    [...]
```

Listing 6.71 "some_playbook.yml": Example Playbook Excerpt

6.14.2 Reading Environment Variables

If you want to read an existing environment variable, the solution depends on *where* the environment variable is located. If it's located on the control host, you can usually read it very easily with an `env` lookup. (If you need to, reread Section 6.8).

However, if the environment variable lives on a target host, you can only easily read it if you access your target user directly with Ansible (i.e., not via `ansible_become`) or if the variable is set globally (for example, via */etc/environment*). In those cases, your variable should be present in the `ansible_env` fact:

```
---
- hosts: debian

  tasks:
    - debug: var=ansible_env.ANIMAL
```

Listing 6.72 "read_env.yml": Reading Environment Variable on Target Host

```
[…]
ok: [debian] =>
  ansible_env.ANIMAL: Cat
[…]
```

Important

Keep a potentially active fact cache in mind! If you're concerned that the cache information is outdated, you should ensure that there are fresh facts beforehand (Section 6.3.2).

However, reading the variable becomes rather difficult when you access your target user via privilege escalation (i.e., `ansible_become`) and the variable is defined in a shell

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

startup file of this user. Then, you generally have to access the variable "manually," like this:

```
---
- hosts: debian

  tasks:
    - command: bash -lc 'echo $ANIMAL'
      changed_when: false
      register: bash_cmd

    - debug: var=bash_cmd.stdout
```

Listing 6.73 "read_become_user_env.yml": Reading Environment Variable on Target Host

6.14.3 Environment Variables in "path" or "dest" Attributes

If you're reading environment variables via a simple lookup in `ansible_env`, you can also use these variables in remote path attributes of common file modules:

```
---
- hosts: debian

  tasks:
    - file:
        path: /tmp/${ANIMAL}.txt
        state: touch

    - copy:
        dest: /tmp/${ANIMAL}.txt
        content: |
          Some content...

    - lineinfile:
        path: /tmp/${ANIMAL}.txt
        line: More content...
```

Listing 6.74 "use_envar_in_path.yml": Using Environment Variables in Remote Path Attributes

This is because these path specifications are sent through the `os.path.expandvars()` and `os.path.expanduser()` Python methods before they are actually used. The former method is responsible for variable expansion, and the latter method is responsible for tilde (~) expansion, which is why it also works in path specifications.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 7
Using Modules and Collections

Modules are the workhorses of Ansible. With them, you can reliably accomplish well-defined goals through *tasks*. This is also part of the well-known UNIX philosophy: tools that can master one thing well are essential. Of course, there are some modules that can do too much, but such exceptions only prove the rule.

In this chapter, I give you an initial overview of modules you will *typically* need quite often. You have already seen many of them in the previous chapters. But to create a basis for understanding, we must first discuss collections, which are used to distribute modules, among other things.

7.1 Collections

Collections are Ansible's standardized distribution format that enables additional components like playbooks, modules, roles, and plug-ins to be packaged and shared with users. They constitute a relatively new Ansible concept that was fully implemented with version 2.10.

One of the reasons for the introduction of collections was that the old development concepts of Ansible ("Everything you need is included" and "Batteries included") were no longer sustainable. With more than 3,000 modules available at that time, it became increasingly difficult to achieve reasonable releases, most of which were maintained by the community.

Thus, the core of Ansible (*ansible-core*) was separated from the modules and plug-ins contributed by the community into different development paths. (We discussed this in depth in Chapter 1, Section 1.3.)

7.1.1 A Minimal Environment with "ansible-core"

Let's take a look at this in practice. You should use a virtual environment so you can do this risk-free without interfering with a live installation.

First, you should set up a virtual environment (see also Chapter 1, Section 1.6)—and we emphasize again here that you do not need root privileges for this:

```
$ python3 -m venv ~/venv/ansible-core
$ source ~/venv/ansible-core/bin/activate
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Now, you can install the latest stable *ansible-core* package that is compatible with your Python version:

```
$ pip3 install ansible-core
```

It will finish fairly quickly, and one reason for that is that instead of several thousand modules, we now only have a few dozen important base modules on board:

```
$ ansible-doc -l | wc -l
70
```

If you are curious, feel free to omit `| wc -l` and examine the list. It includes many old acquaintances, but you may also notice at second glance that many of your favorite modules are missing.

An example of a module that is included in a "normal" Ansible installation but not here is `ini_file`:

```
$ ansible-doc ini_file
[WARNING]: module ini_file not found in:
[...]
```

This module is part of the `community.general` collection. It's documented at *https://docs.ansible.com/ansible/latest/collections/community/general/ini_file_module.html*, and you can easily find it via a search.

7.1.2 Managing Collections

In the following sections, we'll walk through installing collections, listing the installed collections, and extending the collection search path.

Installing Collections

With the `ansible-galaxy` command, you can install collections from the Internet, among other things. This is the simplest invocation for that:

```
$ ansible-galaxy collection install <COLLECTION_NAME>
```

We assume that we are still inside the activated virtual environment, but a normal collection installation would write to *~/.ansible/collections*, which might later affect our live Ansible. To avoid this, you create a new directory like this one:

```
$ mkdir -p ~/coredemo/collections
```

You can now set this directory as the installation target:

```
$ ansible-galaxy collection install \
    -p ~/coredemo/collections community.general
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

To ensure that Ansible considers this location when searching for collections, you can (in our case temporarily) work with the `ANSIBLE_COLLECTIONS_PATH` environment variable:

```
$ export ANSIBLE_COLLECTIONS_PATH=~/coredemo/collections
```

The module `ini_file` (and many others) will then be available again:

```
$ ansible-doc ini_file

$ ansible-doc -l | wc -l
659
```

Listing Installed Collections

You can use the following code to view a summary of all collections installed, along with their respective versions:

```
$ ansible-galaxy collection list
```

Extending the Collection Search Path via Configuration

Instead of the `ANSIBLE_COLLECTIONS_PATH` environment variable, you can of course use a the aptly named `collections_path` configuration parameter. (However, the environment variable is typically "stronger"—so be careful if you use both!)

Unfortunately, there is no simple way to *extend* the search path. You need to find out the current state and add your path or paths. The Ansible default is typically `~/.ansible/collections:/usr/share/ansible/collections`, but it's better that you verify this. For example, you can do it this way:

```
$ ansible-config dump | grep COLLECTIONS_PATH
```

If you want to extend the search path in your project with the *~/coredemo/collections* folder, the configuration might look like this:

```
# [defaults]
collections_path = [... existing search paths ...]:~/coredemo/collections
```

Listing 7.1 "ansible.cfg": Extending Collection Search Path

Note

Please do not do this in the lab now; instead, you can exit the virtual environment with `deactivate` (in case you followed this little detour).

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

7.1.3 The Fully Qualified Collection Name

Especially if you have been working with Ansible for a while, you may have noticed that the documentation now consistently uses fully qualified module names. It does it like this:

```
- community.general.ini_file:
    path: /tmp/test.ini
    […]
```

And it did it before like this:

```
- ini_file:
    path: /tmp/test.ini
    […]
```

Ansible implements a typical namespace concept, which primarily avoids name collisions. Theoretically, there could be another collection that also contains a module named `ini_file`.

Currently, many Ansible users (myself included) still use the short form for "standard modules," even though the official documentation suggests not doing so anymore. The simple reason is that the short form still works and the long form (apart from more typing) does not add real value.

The Ansible developers naturally had to support this; otherwise, a lot of people would have had to change thousands of tasks in hundreds of playbooks and roles, which would have annoyed many of them. Therefore, it was decided that all modules, plugins, and other objects included in the old Ansible up until the transition cut-off date will remain accessible in the short form going forward.

If you'd like to see how this is done and you've already set up the virtual environment from the previous section, try the following command:

```
$ grep -r ini_file ~/venv/ansible-core
```

You'll discover an *ansible_builtin_runtime.yml* file that contains a multitude of aliases (short name → long name).

However, even proponents of the short form will occasionally use fully qualified names, either when using modules that did not exist in the old days or when they have changed their names since then. An example of the second case is the `community.crypto.x509_certificate` module for certificate management. In the old days, it was called `openssl_certificate` and could indeed still be referred to by this name. However, the new name is only available in long format, and to avoid confusion, you should always refer to Ansible objects by their current name.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

7.1.4 Interim Conclusion

As long as you are using an Ansible community package and are satisfied with its contents, you don't need to worry too much about collections. However, we have seen that even installing collections later wouldn't be a problem.

On the other hand, if you ever want to develop your own Ansible components like modules or plug-ins *and* distribute them in a standardized format, this topic becomes considerably more interesting. We will come back to it later.

And now, we finally come to the main focus of this chapter: the modules.

7.2 Module

Depending on the version, an Ansible community package contains from 6,000 to 8,000 modules. You can get a list of all locally available modules with the following code:

```
$ ansible-doc -l
```

You can also read the module offline, as known from the web, by using this:

```
$ ansible-doc <MODULE_NAME>
```

Due to restructuring with collections, there is no longer a single overview page online. However, you can find a fairly comprehensive index at *https://docs. ansible.com/ansible/latest/collections/*.

As long as it is still available, you can also use the module index of version 2.9, which is grouped by categories quite clearly at *https://docs.ansible.com/ansible/2.9/modules/modules_by_category.html*. However, the information contained there is now quite outdated, so I can no longer recommend it unreservedly. In general, you should always make sure that the online documentation you are reading matches the version of Ansible you are using!

In the following sections, I would like to present you with a selection of modules and provide one or two examples of how to call each one of them: one fairly short example and (if applicable) a second one that demonstrates more advanced calling options. I will only mention *some* modules, but in any case, the rule is this: if a module is interesting or important to you, if you have any doubts about it, you should consult the official documentation.

If a module is not from the core or `ansible.builtin`, I will use its fully qualified collection name (FQCN). If it's also not included in the Ansible community package, I will specifically point that out.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

7.3 Modules for Command Execution

The following modules are available for command execution.

7.3.1 "command": Executing Commands

```
- name: Search for config files and store result
  command: find /etc -name *.conf
  register: find_cmd

- name: Store sorting result in /tmp
  command:
    chdir: /tmp
    cmd: sort /etc/passwd -o sorted.txt
    creates: sorted.txt
```

The second example shows how to change the working directory with `chdir` and an "idempotency marker" with `creates`.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/command_module.html*.

7.3.2 "shell": Executing Commands via a Shell

```
- name: Count programs that start with "a"
  shell: "ls -l /usr/bin/a* | wc -l"

- name: Brace-Expansion (requires Bash)
  shell:
    cmd: echo test{1,2,3}
    executable: /bin/bash
```

This module is similar to the `command` module, except that the command is executed via `/bin/sh` by default. If you need shell mechanisms that are not supported by a classic Bourne shell, you can also specify the desired shell interpreter (see the second example).

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/shell_module.html*.

7.3.3 "raw": Executing Commands without the Python Subsystem

```
- name: Install Python 3 on Debian-like target host
  raw: apt-get -y install python3
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

This module is typically used to install a Python package on a target system that does not have Python installed. From a technical perspective, it's comparable to the `shell` module; the `executable` parameter is also available here.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/raw_module.html*.

7.3.4 "script": Transferring and Executing Local Scripts

```
- name: Transfer and run script
  script: /usr/local/bin/myscript.sh --opt1 1234
```

The name of this module is somewhat misleading, as you can use many types of programs. Of course, in practice, they will mostly be Shell, Perl, or Python scripts. It is also interesting that (just like with the `raw` module) Python is not required on the target machine. Apart from that, the parameterization options are similar to those of the `shell` module.

The online documentation of this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/script_module.html*.

7.4 Modules for Package Management

The following modules are used for package management.

7.4.1 "apt": Package Management on Debian/Ubuntu Systems

```
- name: Update package index (like apt update)
  apt:
    update_cache: yes
    cache_valid_time: 3600

- name: Install package tmux
  apt:
    name: tmux
```

The `apt_repository` and `apt_key` modules are available for managing repositories and keys.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/apt_module.html*.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

7.4.2 "dnf" and "yum": Package Management on Rocky, CentOS, Red Hat, and Fedora Systems

```
- name: Install package tree
  dnf:
    name: tree

- name: Install package tmux
  yum:
    name: tmux
```

The `yum_repository` module is available for managing repositories.

The online documentation for it is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/dnf_module.html*.

7.4.3 "community.general.zypper": Package Management on SUSE Systems

```
- name: Install package tmux
  zypper:
    name: tmux
```

The `zypper_repository` module is available for managing repositories.

The online documentation for it is at *https://docs.ansible.com/ansible/latest/collections/community/general/zypper_module.html*.

7.4.4 "package": Generic Package Management

```
- name: Install package tmux
  package:
    name: tmux
```

The `package` module is the lowest common denominator of package management. Under the hood, it simply calls the actual module (`apt`, `dnf`, etc.).

The online documentation for it is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/package_module.html*.

7.4.5 "package_facts": Representing Package Information as Facts

```
- name: Collect package information
  package_facts:

- block:
    - debug: msg="Apache is installed."
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    - debug: var=ansible_facts['packages']['apache2']
  when: ansible_facts['packages']['apache2'] is defined
```

The `package_facts` module integrates information about installed distribution packages into the facts. For the above case, the data structure per package looks like this:

```
"ansible_facts['packages']['apache2']": [
        {
            "arch": "amd64",
            "category": "httpd",
            "name": "apache2",
            "origin": "Debian",
            "source": "apt",
            "version": "2.4.62-1~deb12u1"
        }
    ]
```

The online documentation for it is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/package_facts_module.html*.

7.4.6 Some Other (Not Further Described) Modules

Other modules you may use include the following:

- You can use `pip` to manage Python modules.
- You can use `community.general.pear` to manage PHP packages.
- You can use `community.general.cpanm` to manage Perl modules.

7.5 Modules for Managing Files and File Contents

With the following modules, you can manage both files and file contents.

7.5.1 "copy": Copying and Uploading Files

```
- name: Upload test1.txt
  copy:
    src: test1.txt
    dest: /tmp

- name: Upload test1.txt and set permissions
  copy:
    src: test1.txt
    dest: /tmp
    owner: user1
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    group: users
    mode: 0600
```

If the source file is specified by a relative path, Ansible searches in a *files/* folder alongside the playbook and then directly at the location of the playbook. When the source file is used in a role, Ansible first searches in the role's *files/* folder. If the source file to be copied is on the target system, then you indicate this with the parameter `remote_src: yes`.

You can specify the `mode` numerically or symbolically (e.g., `'u+rw'`). Additionally, you can use the `preserve` keyword, which allows the target file to have the same permissions as the source file.

Attention

There are two correct ways to specify a numeric mode:

- Use a number with a leading 0 (as in the example above) so that Ansible's YAML parser will recognize an octal number.
- Use a *quoted* string like `'644'` or `'1777'` to have Ansible convert it internally.

If you specify a number in a different way, it will not work correctly!

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/copy_module.html*.

7.5.2 "template": Processing Files with Jinja and Uploading

```
- name: Process template and upload result
  template:
    src: index.html.j2
    dest: /srv/www/htdocs/index.html
```

This module usage is almost identical to `copy`, except that the input is first processed by the Jinja engine. (If the term *Jinja* is unfamiliar to you, you should read Chapter 6, Section 6.6.) Please note that for relative path specifications, Ansible also searches for the source file in the *templates/* folder (and *not* in *files/*!).

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/template_module.html*.

7.5.3 "file": Managing Files and File Attributes

```
- name: Set file permissions
  file:
    path: /etc/ssl/private
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    owner: root
    group: ssl-cert
    mode: 0710
```

With `file`, you can create or delete files and directories, manage soft and hard links, and much more.

Creating links with this module is not always straightforward, so here are some notes. Suppose you want to solve with Ansible what you would typically solve in the shell like this:

```
$ ln -s <TARGET_NAME> <LINK_NAME>
```

The corresponding Ansible task looks like this:

```
- name: Create symlink
  file:
    src:  TARGET_NAME
    dest: LINK_NAME
    state: link
```

The result would be a hard link with `state: hard`. That is problematic in two situations:

- The old file or the source does not exist (which would not be a problem for the `ln` command with symlinks).
- The new name (i.e., the link path) already exists but is not a symlink (rather, it's a directory, a regular file, or whatever).

Ansible would consider all these cases as errors, but you can avoid this with the additional `force: yes` module parameter.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/file_module.html*.

7.5.4 "stat": Obtaining Information about Files

```
- name: Retrieve file status
  stat:
    path: /etc/ssl
  register: p

- debug:
    msg: /etc/ssl is an existing directory
  when: p.stat.exists and p.stat.isdir
```

Here's a tip: take a look at the full structure of the registered variables (`debug: var=p`). There is a lot more information available about this in the online documentation at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/stat_module.html*.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

7.5.5 "lineinfile": Managing Lines in Text Files

```
- name: Change line
  lineinfile:
    path: /etc/selinux/config
    regexp: '^SELINUX='
    line: 'SELINUX=disabled'

- name: Insert line
  lineinfile:
    path: /etc/ssh/sshd_config
    insertafter: '^#ListenAddress'
    regexp: '^ListenAddress'
    line: 'ListenAddress 192.168.150.20'
```

To successfully use this module (as well as the `blockinfile` and `replace` modules), you need to be reasonably proficient in handling regular expressions. If you need to catch up here or need an introduction, then take a look at Appendix D. To learn about the intricacies of the specific Ansible "dialect" that's in use here, please visit *http://docs.python.org/3/library/re.html*.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/lineinfile_module.html*.

7.5.6 "blockinfile": Managing Text Passages in Files

```
- name: Insert virtual interface configuration
  blockinfile:
    path: /etc/network/interfaces
    block: |
      iface ens33:0 inet static
          address 10.0.0.1/24
          netmask 255.255.255.0

- name: Insert first virtual interface configuration
  blockinfile:
    path: /etc/network/interfaces
    marker: "# {mark} ANSIBLE MANAGED BLOCK: Virtual interface 1"
    block: |
      iface ens33:0 inet static
          address 10.0.0.1/24
          netmask 255.255.255.0
```

The module manages text passages surrounded by configurable marker lines. Options are similar to those for positioning with `lineinfile`.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

If you want to manage different text passages within the same file, you must take care to work with unique markers, as the second example demonstrates. Ansible uses the resulting corresponding `BEGIN`/`END` comments as references.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/blockinfile_module.html*.

7.5.7 "replace": Searching and Replacing in Text Files

```
- name: Deactivate PasswordAuthentication
  replace:
    path: /etc/ssh/sshd_config
    regexp: '^#?(PasswordAuthentication).*'
    replace: '\1 no'
```

The module replaces *all* occurrences of `regexp` in the file with the `replace` expression. If `replace` is missing, the occurrences are deleted.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/replace_module.html*.

7.5.8 "unarchive": Uploading and Extracting Archives

```
- name: Upload archive and extract in /srv/www/webapps
  unarchive:
    src: 'roundcubemail-1.3.4-complete.tar.gz'
    dest: /srv/www/webapps
    owner: www-data
    group: www-data
    creates: '/srv/www/webapps/roundcubemail-1.3.4'
```

This module can handle the *.zip*, *.tar*, *.tar.gz*, *.tar.bz2*, and *.tar.xz* archive formats. If the source is specified as a relative path, then the search strategy is the same as with the `copy` module. The target directory must exist.

Just as with the `copy` module, the `remote_src: yes` parameter is also available here, if the archive file is already on the target system.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/unarchive_module.html*.

7.5.9 Some Other (Not Further Described) Modules

Other modules you may use include the following:

- You can use `ansible.posix.acl` to manage file ACLs.
- You can use `assemble` to assemble file fragments.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

- You can use `fetch` like `copy`, but in reverse.
- You can use `find` to use the `find` command as a module.
- You can use `community.general.ini_file` to manage contents of INI files.
- You can use `community.general.pamd` to manage PAM configuration.
- You can use `ansible.posix.synchronize` to use the `rsync` command as a module.
- You can use `tempfile` to create temporary files or directories.
- You can use `community.general.xml` to manage contents of XML files.

7.6 Modules for Other Typical Administrative Tasks

You can use the following modules for other typical administrative tasks:

7.6.1 "service": Starting, Stopping, and Restarting Services, Etc.

```
- name: Start service and enable start on boot
  service:
    name: httpd
    state: started
    enabled: yes
```

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/service_module.html*.

7.6.2 "service_facts": Presenting Service Information as Facts

```
- name: Collect service facts
  service_facts:

- debug: msg="Apache is installed and running"
  when: >
    ansible_facts['services']['apache2'] is defined
      and
    ansible_facts['services']['apache2']['state'] == 'running'

- debug: var=ansible_facts['services']['apache2']
```

The `service_facts` module assembles information about installed distribution packages into facts. For the above case, the data structure per service looks like this:

```
    "ansible_facts['services']['apache2']": {
        "name": "apache2",
        "source": "sysv",
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
        "state": "running"
    }
```

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/service_facts_module.html*.

7.6.3 "cron": Managing Cron Jobs

```
- name: Setup cronjob in /etc/cron.d/
  cron:
    cron_file: certbot
    user: root
    name: Run "certbot renew" once per week
    special_time: weekly
    job: certbot renew --post-hook "systemctl restart postfix apache2"
```

Of course, parameters like `minute`, `hour`, and `day` are also valid time specifications.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/cron_module.html*.

7.6.4 "hostname": Changing the Hostname

```
- name: Set hostname to "web01"
  hostname:
    name: web01
```

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/hostname_module.html*.

7.6.5 "user" and "group": User and Group Management

```
- name: Create group "vmail"
  group:
    name: vmail
    gid: 5000

- name: Create user "vmail"
  user:
    name: vmail
    uid: 5000
    group: vmail
    home: /srv/imap
    shell: /bin/false
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

The online documentation for this is on these two websites:

- *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/user_module.html*
- *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/group_module.html*

7.6.6 "ansible.posix.firewalld": Managing firewalld-Based Firewalls

```
- name: Open HTTP port 80/tcp
  firewalld:
    service: http
    state: enabled
    permanent: yes
    immediate: yes
```

This module requires you to have the appropriate Python bindings for *firewalld* management on the target systems. In rare cases, you may need to install the *python3-firewall* package beforehand.

However, be aware that without a solid understanding of *firewalld*, you will not get very far with this module!

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/posix/firewalld_module.html*.

7.6.7 "reboot": Rebooting Machines

```
- name: Restart machine
  reboot:
```

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/reboot_module.html*.

7.6.8 Some Other (Not Further Described) Modules

Other modules you may use include the following:

- You can use `ansible.posix.authorized_key` to manage SSH keys.
- You can use `community.general.filesystem` to create filesystems.
- You can use `getent` as a wrapper for the Linux `getent` command.
- You can use `community.general.locale_gen` to manage locales.
- You can use `community.general.lvg`, `.lvol` to manage LVM devices and volume groups or logical volumes.
- You can use `ansible.posix.mount` to manage mounts.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

7.7 Modules for Interacting with Network Services

The following modules are useful for interacting with network services.

7.7.1 "get_url": Downloading Resources

```
- name: Download penguin image
  get_url:
    url: https://upload.wikimedia.org/wikipedia/commons/a/af/Tux.png
    dest: /tmp/tux.png
```

With this module, you can download an HTTP, HTTPS, or FTP resource to a target system. The system naturally requires direct access to this resource, but you can use proxies by defining the environment variables `<protocol>_proxy` (see Chapter 6, Section 6.14 if necessary).

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/get_url_module.html*.

7.7.2 "uri": Communicating with Web Services

```
- name: Retrieve a Chuck Norris joke
  uri:
    url: https://api.chucknorris.io/jokes/random
    return_content: yes
  register: this

- debug: var=this.json.value
```

Besides such simple queries, the `uri` module offers a wide range of capabilities: POST requests, authentication, cookies, etc.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/uri_module.html*.

7.7.3 "git": Interacting with Git Servers

```
- name: Clone example repository
  git:
    repo: https://github.com/ansible-buch/docker-installer.git
    dest: /tmp/ansible-buch/docker-installer
```

A prerequisite for this operation is the presence of the Git software on the respective systems. Therefore, you may first need to install the *git* package in a preceding task.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Otherwise, the `git` module offers many more possibilities, such as SSH key authentication and checking out branches (with the `version` parameter).

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/git_module.html*.

7.8 Special Modules (Control Flow Management, Etc.)

In this section, you will find a selection of modules that allow you to trigger very specific (mostly internal) actions.

7.8.1 "meta": Triggering Various Ansible Actions

```
- name: Quit play
  meta: end_play
```

The number of available meta-actions has grown steadily over the years. Ansible 2.12 and later know the following nine actions:

- **`clear_facts`**
  This deletes facts and the facts cache.
- **`clear_host_errors`**
  This is very misleading—it merely brings unreachable hosts back on board, and only if there is still a host left.
- **`end_host`**
  This ends the play for the current host (without error).
- **`end_play`**
  This ends the play for all hosts (without error).

  You should not associate `end_play` with *host-specific* conditions (e.g., `when: inventory_hostname == "server27"`). If you do, the behavior will become quite unpredictable and you will not achieve what you intended. You should use `end_host` for this purpose!
- **`flush_handlers`**
  This causes "premature" invocation of handlers; see Chapter 5, Section 5.9.2.
- **`noop`**
  This does nothing (which is necessary sometimes).
- **`refresh_inventory`**
  This triggers a reload of the inventory (which only makes sense with dynamic inventories).
- **`reset_connection`**
  This discards the persistent SSH connections so that any subsequent tasks will establish new connections.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

- `end_batch`
  This ends the current batch. With `serial: 0` or undefined (see Chapter 14, Section 14.2.2), the effect is identical to `end_play`.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/meta_module.html*.

7.8.2 "debug": Generating Output during Playbook Runs

```
- debug: msg="Hello"

- debug: var=ansible_default_ipv4
```

The naming of these tasks with `name` is often omitted, if only because a `debug` task often should not remain in the play. Using the `var` parameter helps when you want to show the contents of a variable easily.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/debug_module.html*.

7.8.3 "fail": Triggering a Failure

```
- set_fact:
    hour: "{{ lookup('pipe', 'date +%H') }}"

- fail:
    msg: "It is simply too late today."
  when: hour|int > 20
```

With the `fail` module, you can deliberately trigger a failure and thus end the play for the respective hosts. In the old days of Ansible, before `meta: end_play` was invented, this was the only way to exit a play.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/fail_module.html*.

7.8.4 "pause": Pausing and Optionally Reading Input

```
- pause:
    prompt: Please turn off the lights now

- name: Give the CPU time to cool down
  pause:
    seconds: 30
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
- pause:
    prompt: Please enter your name
  register: this

- debug: msg="Goodbye {{this.user_input}}!"
```

With the `pause` module, you can stop the execution of a playbook for a period of time.

Attention

Please *do not* use `pause` to wait for interactive input if your playbook might run in non-interactive environments like Cron or Ansible AWX/Tower!

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/pause_module.html*.

7.8.5 "wait_for": Waiting for Certain Events

```
- name: Wait until the file /tmp/foo is present
  wait_for:
    path: /tmp/foo

- name: Wait 300 seconds for port 8000 to become open
  wait_for:
    port: 8000
```

With the `wait_for` module, you can wait in the play until a certain condition is met. The maximum wait time is 300 seconds by default (with the `timeout` parameter). In addition to the possibilities shown here, you can wait for a port to close or you can compare file contents or sockets with a regular expression and wait until the pattern matches.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/wait_for_module.html*.

7.8.6 "wait_for_connection": Waiting Until a Target Host Is Reachable

```
- name: Wait 600 seconds for target connection to become reachable/usable
  wait_for_connection:
    timeout: 600
```

The timeout of 600 seconds is the default and therefore does not need to be specified separately.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/wait_for_connection_module.html*.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

7.8.7 "assert": Ensuring that Certain Conditions Are Met

```
- name: It must be Debian 9 or 10
  assert:
    that:
      - ansible_distribution == "Debian"
      - ansible_distribution_major_version is version('9', '>=')
      - ansible_distribution_major_version is version('10', '<=')
    fail_msg: "Only Debian version 9 or 10 is supported"
```

The `assert` module ensures that all specified conditions are met; otherwise, processing is aborted with an error. We already discussed this in detail in Chapter 6, Section 6.1.10.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/assert_module.html*.

7.8.8 "set_fact": Setting Variables during Runtime

```
- name: Set two variables
  set_fact:
    foo: 100
    bar: 200

- debug: msg="foo = {{foo}}, bar = {{bar}}"
```

With `set_fact`, you can set host-specific variables at runtime. If you need to, read Chapter 6, Section 6.1.4.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/set_fact_module.html*.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 8
Modularization with Roles and Includes

If you were to configure real servers by using the methods we've discussed up to now, you would quickly have playbooks that contain several hundred tasks. However, this is not desirable for purposes of clarity, maintainability, or reusability.

8.1 Creating and Using Roles

To address this issue, I would now like to introduce you to *roles*. In Ansible, a role is the primary mechanism you use to organize a playbook's tasks thematically so as to achieve reusability.

8.1.1 The Concept of Roles in Ansible

In Ansible, each role has a name that directly maps to a directory. Here are a couple of examples of meaningful names:

- *apache*
- *debian/openldap*

How much detail you put into the organization of your role hierarchies is a matter of personal preference, or it can depend on the specific problem you're addressing. Usually, you should prefer flatter hierarchies or no hierarchy at all.

Note

When choosing the role name, you should limit yourself to ASCII letters, numbers, underscores, and directory separators (i.e., characters from the `[a-zA-Z0-9_/]` set).

You can easily use other characters like the period (`'.'`) or the dash (`'-'`), but if you want to migrate your role into a collection one day, these will no longer be allowed.

Search Paths

You can search for roles in the *roles/* folder of the *respective* playbook (and in a few other places that are not relevant to our project).

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

To keep things clear, I recommend using a *roles/* folder alongside the *playbooks/* folder instead:

```
$ cd ~/ansible/projects/start; mkdir roles
```

To have Ansible search for roles there as well, you need to extend the role search path. For this, you use the `roles_path` configuration directive. You can also specify more than one directory by simply separating them with a colon. You should therefore use the following configuration:

```
# [defaults]
roles_path = ./roles
```

Listing 8.1 "ansible.cfg": Extending Role Search Path

The default search paths remain intact, and if Ansible doesn't find a role, it will also print the search paths in the error message. This should help you resolve any issues quickly.

Structures of Roles

All files belonging to a role are structured into certain subdirectories. Here is an overview:

- *<ROLE>/***tasks***/main.yml*
  This consists of the tasks of the role.
- *<ROLE>/***files***/**
  These are the files to upload.
- *<ROLE>/***templates***/**
  These are the Jinja templates to upload.
- *<ROLE>/***handlers***/main.yml*
  These are the handlers that belong to the role.
- *<ROLE>/***defaults***/main.yml*
  These are the variables with very low priority (Rank 2 in our list from Chapter 6, Section 6.1.3).
- *<ROLE>/***vars***/main.yml*
  These are the variables with very high priority (Rank 15 in our list from Chapter 6, Section 6.1.3).
- *<ROLE>/***meta***/main.yml*
  This is meta information (e.g., dependencies on other roles).

Note

You can omit unnecessary directories. For example, if your role does not use templates, then you do not need the subfolder *templates/*.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

8.1.2 A Simple Example of a Role

To make all of this a bit more concrete, we will now create a first, very simple role. For simplicity's sake, let's call it `hello`. Starting from the root folder of our project, you will need the following structure:

```
roles/
`-- hello/
    `-- tasks/
        `-- main.yml
```

Please create the directory hierarchy below *roles/* accordingly. We also keep the content of the *main.yml* file very simple:

```
---
- name: Say hello
  debug: msg="Hello from a role!"
```

Listing 8.2 "hello/tasks/main.yml": Task in Role

Usually, to load and start a role, you would create a corresponding playbook—and I will discuss that in the next section. However, you can also start a role ad hoc with an `import_role` call:

```
$ ansible localhost -m import_role -a name=hello
localhost | SUCCESS => {
    "msg": "Hello from a role!"
}
```

8.1.3 Using Roles in a Playbook

If you have a role available and want to use it, you'll typically call it in a playbook. There are two basic syntactic variants available to you here. The one you already learned in the ad hoc command also works in the playbook, as follows:

```
---
- hosts: all

  tasks:
    - import_role:
        name: hello
```

Listing 8.3 "hello-role_v1.yml": Embedding Role in Playbook

(Instead of `import_role`, you can also use `include_role`; we'll discuss the differences between `import_*` and `include_*` in Section 8.3.1.)

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

The second variant is more common when you're working with roles exclusively. Here, you replace `tasks` with `roles`:

```
---
- hosts: all

  roles:
    - name: hello
```

Listing 8.4 "hello-role_v2.yml": Embedding Role in Playbook

And instead of the attribute `name`, you can use `role`:

```
---
- hosts: all

  roles:
    - role: hello
```

Listing 8.5 "hello-role_v3.yml": Integrating Role into Playbook

Or you could omit the attribute altogether (which is not recommended):

```
---
- hosts: all

  roles:
    - hello
```

Listing 8.6 "helo-rolle_v4.yml": Integrating Role into Playbook

It is also very useful to supply a role with parameters (variables) when it is called. These will then only be visible within the role:

```
  roles:
    - name: hello
      color: blue
      number: 57
```

Listing 8.7 Example of Role Parameterization

If you prefer the variant of calling roles by using `import_role`, the parameterization will look like this:

```
  tasks:
    - import_role:
        name: hello
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    vars:
      color: blue
      number: 57
```

Listing 8.8 Parameterization of Role When Using "import_role"

Do you still remember the precedence rules from Chapter 6, Section 6.1.3? With this method of parameterization, you would be at rank 20, which is hard to beat.

8.1.4 Plays with Roles and Tasks, "pre_tasks" and "post_tasks"

In a play(book), you can certainly use roles *and* simple tasks together:

```
roles:
  [...]

tasks:
  [...]
```

However, remember that you are using a map here, and as such, it has no order, from a technical perspective! So, it doesn't matter whether you list your `roles` first and your `tasks` second in the playbook or vice versa! Only Ansible's internal handling matters:

- First, all roles inside `roles` are executed.
- Then, all tasks inside `tasks` are executed.

If you absolutely want or need to mix roles and tasks, then you should at least make the order immediately clear to all readers of the playbook. We therefore recommend that you either use the `import_role` variant (in which you would have tasks only) or specify your tasks below a `pre_tasks` or `post_tasks` section of the play. This achieves what the name already suggests: an execution of these tasks as early or as late as possible.

If Debian systems are involved in a play, you'll often see the update of the software directory as a pre-task:

```
pre_tasks:
  - name: Update apt cache if needed
    apt: update_cache=yes cache_valid_time=3600
    tags: always
```

Listing 8.9 Example of Using "pre_tasks"

8.1.5 Dependencies between Roles

A role (e.g., X) can define dependencies on other roles (e.g., Y and Z). When you apply X, then Y and Z would automatically be executed *first*.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Technically, you do this through a *main.yml* file in the *meta/* role directory:

```
---
dependencies:
  - Y
  - Z
```

Listing 8.10 "X/meta/main.yml": Meta Information about Role

You could also parameterize the base roles:

```
---
dependencies:
  - role: Y
    var1: value1
    var2: value2

  - role: Z
    var3: value3
```

Listing 8.11 "X/meta/main.yml": Parameterization of Base Roles

The whole thing is implemented quite intelligently: if two different roles use the same base role, it is only called once—*if* the parameterization is identical!

Rating

From a technical perspective, using role dependencies is a great idea. Role A automatically calls B and C, and as a result, B automatically calls D and E first, and so on. However, when you're actually working with them, you will often find yourself sitting there with your mouth open, wondering "Why did it do that now?" This will happen even if you set all of them up yourself just three days ago.

But feel free to try it out and prove me wrong. In any case, I do not want to make a definite recommendation for or against using this feature.

8.1.6 Choosing Other Startup Files

For the sake of completeness, I should mention that instead of *main.yml*, you can use a different file name as an entry point into the subfolder of a role, even though you will rarely need it.

However, you must call the role with `import_role` or `include_role`, and then, the `tasks_from`, `handlers_from`, `defaults_from`, and `vars_from` parameters will be available to you.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

8.1.7 Creating New Roles with "ansible-galaxy"

Creating a directory structure for a new role is a bit tedious and somewhat error-prone. (For example, making a typo like typing *handler* instead of *handlers* when creating one of the directories is enough to break it.)

Therefore, you should let the `ansible-galaxy` command assist you to prevent errors. You should switch to the directory where your new role should be created (in our case *roles/*) and call the following:

```
$ ansible-galaxy role init <ROLENAME>
```

Or, you can use a shorter one ("`role`" is the default subcommand):

```
$ ansible-galaxy init <ROLENAME>
```

Try it out:

```
$ ansible-galaxy init testrole
- Role testrole was created successfully

$ tree testrole
testrole
|-- defaults
|   `-- main.yml
|-- files
|-- handlers
|   `-- main.yml
|-- meta
|   `-- main.yml
|-- README.md
|-- tasks
|   `-- main.yml
|-- templates
|-- tests
|   |-- inventory
|   `-- test.yml
`-- vars
    `-- main.yml
```

This is more than complete, and you should feel free to delete folders you do not need. First and foremost, the *meta/* folder is useful when you're working with role dependencies or when you want to publish your role in Ansible Galaxy. You'll also need the *tests/* folder when you use CI systems like Jenkins and Travis CI.

The `ansible-galaxy role init` command also has a few options. At this point you should know these:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

- With the `--force` option, you can overwrite an existing role.
- With the `--init-path <PATH>` option, you can choose a target folder that's different from the current directory.
- With the `--role-skeleton <PATH>` option, you can use another template to initialize the role.

8.2 The Online Ansible Galaxy Repository

The *Ansible Galaxy* is *the* online repository for roles and collections. You can find the project homepage at *https://galaxy.ansible.com/*.

You can also browse roles by selecting **Roles** in the navigation area on the left and then selecting **Roles** again. Then, you can enter a search term such as `postgres` in the search field.

When you click on a promising role, you'll go to its page. The most important information should be under the **Documentation** tab.

If you want to delve into the implementation, you will usually find a link to the **GitHub Repo** at the top right. There, you can peruse the source files at your leisure.

When you've found what you're looking for, you can also download roles directly from the Galaxy with this:

```
$ ansible-galaxy install <author.rolename>
```

(This assumes your control host has direct internet access, of course.)

Attention

As you can imagine, it's not advisable to simply adopt and apply just any Galaxy roles without prior review. You can, of course, be a bit more relaxed about this with authors you trust or prominent community collections.

8.3 Using Imports and Includes

In addition to roles, Ansible offers a simpler concept for modularization: you can incorporate tasks, variables, and even complete playbooks from external files with an import or include. This is often utilized to increase clarity and reusability.

8.3.1 "import_tasks" and "include_tasks"

In more complex roles, for instance, the *tasks/main.yml* is often split into several files:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- import_tasks: do_this.yml
- import_tasks: do_that.yml
```

Listing 8.12 "tasks/main.yml": Example Use of Imports

As an alternative to the `import_tasks` directive, you could use `include_tasks`, and the above example would work just as well with it.

So, what's the difference? Simply put, an import happens during parsing time and an include occurs during the runtime of the playbook. Therefore:

- **You use imports for static matters**, which are fixed from the beginning.
- **You use includes for dynamic matters**, which only arise during runtime.

For example, you can only perform repeated inclusion inside a loop or include a file with a dynamic filename with `include_tasks`:

```
- include_tasks: part.yml
  with_items:
    - foo
    - bar

- include_tasks: "{{ ansible_os_family }}.yml"
```

Listing 8.13 Two Examples that Can Only Be Realized with "include_tasks"

On the other hand, if you need to use `ansible-playbook` with options like `--list-tags`, `--list-tasks`, or `--start-at-task`, you must use `import_tasks`.

You can find the details at *https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse.html*, and there is an important takeaway on this page that is worth quoting:

> *Although you can use both dynamic and static reuse in a single playbook, it is best to select one approach per playbook. Mixing static and dynamic reuse can introduce difficult-to-diagnose bugs into your playbooks.*

8.3.2 "include_tasks" and Tags

Using `include_tasks` and tags in combination has some pitfalls. Let's consider the following example:

```
---
- hosts: localhost
  gather_facts: false
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
  tasks:
    - include_tasks: some_tasks.yml
      tags: tasks
```

Listing 8.14 "main-playbook.yml"

And let's consider the file to be included:

```
---
- debug:
    msg: First task
  tags: task1

- debug:
    msg: Second task
  tags: task2
```

Listing 8.15 "some_tasks.yml"

The following call is now up for discussion:

```
$ ansible-playbook main-playbook.yml -t tasks
```

The include itself is executed, but not the included tasks! (By the way, this would work with `import_tasks`, but you don't always have a choice.) However, you might find this entirely intuitive and immediately notice that the following call works as expected:

```
$ ansible-playbook main-playbook.yml -t tasks,task1
```

However, what can you do if the included tasks do not have any tags at all? How can you execute them as a whole? There are two options. The first is to use a block; it might look like a quick-and-dirty option, but it is actually suggested by official sources:

```
---
- hosts: localhost
  gather_facts: false

  tasks:
    - block:
        - include_tasks: some_tasks.yml
      tags: tasks
```

Listing 8.16 "main-playbook2.yml"

The second option relies on the `include_tasks` `apply` parameter to "inherit" various keywords into the included tasks. This process seems considerably more complicated and offers no advantages, at least when it comes to the use case of tags:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- hosts: localhost
  gather_facts: false

  tasks:
    - include_tasks:
        file: some_tasks.yml
        apply:
          tags: tasks
      tags: tasks
```

Listing 8.17 "main-playbook3.yml"

8.3.3 Dynamic Loading of Variables with "include_vars"

The `include_vars` module is intended to let you dynamically reload variables. To handle differences between OS distributions, you'll often use it like this:

```
---
- name: Include OS-specific settings
  include_vars: "{{ ansible_os_family }}.yml"
```

Listing 8.18 "tasks/main.yml": Dynamic Loading of Variable Settings

We've already seen this in Chapter 6, Section 6.5.4. When applied in a role, the search directory for relative path specifications is the *vars/* folder.

If you have only a few OS distribution variants, using an approach like this is very clever:

```
---
- name: Include OS-specific settings
  include_vars: "{{ item }}"
  with_first_found:
    - "{{ ansible_os_family }}.yml"
    - defaults.yml
```

Listing 8.19 "tasks/main.yml": Example of Using "with_first_found" in Role

As the name suggests, `with_first_found` attempts to find and provide the specified files in order. The first match wins, and then, the search is terminated.

8.3.4 "import_playbook"

By using `import_playbook`, you can even include entire playbooks within another playbook:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- import_playbook: playbook1.yml
- import_playbook: playbook2.yml
```

Listing 8.20 "main-playbook.yml": Playbook That Includes Other Playbooks

It is noteworthy that each `import_playbook` statement creates a play. Therefore, you cannot use `import_playbook` at the task level but only at the top play level. However, you could use normal plays (that start with `hosts:`) and `import_playbook` statements one after the other at the same (top) level.

Scopes of Facts and Variables

If facts are set in an early playbook (and thus ultimately in an early play), they will also be available in all later playbooks and plays. This applies to system facts, `set_fact`, and `register`.

However, this does not apply to "normal" variables (e.g., `vars`). Their scope is limited to the corresponding play!

Parameterization

You can even parameterize `import_playbook` statements:

```
---
- import_playbook: playbook1.yml
  vars:
    color:  blue
    number: 100
```

Listing 8.21 "main-playbook2.yml": Playbook that Includes and Parameterizes Another Playbook

8.4 Apache Once Again

Now that we've discussed the theory of all techniques, let's revisit our Apache challenge and transform the playbook from Chapter 6, Section 6.5.5, into a role. For simplicity's sake, let's call it `apache`.

First, you create an empty shell as a starting point in the *roles/* folder:

```
$ cd roles
$ ansible-galaxy init apache
- Role apache was created successfully
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Then, you remove the directories that are unnecessary for this role:

```
$ rm -rf apache/{files,meta,tests}
```

First things first: the tasks. Basically, you carry over the existing task list. (It was fine in the playbook, and of course, it still is) You'll see the more exciting version with the external template file in Listing 8.22. Now, you can also dispense with the `apache_` prefix in the names of the included variable files as follows, as there is no longer any risk of collision in the role context:

```
---
- name: Load system-specific parameters
  include_vars: >
    {{ ansible_distribution | lower | replace(" “: "-") }}.yml

- name: Update package index
  apt:
    update_cache: yes
    cache_valid_time: 3600
  when: ansible_os_family == "Debian"

- name: Install Apache package
  package:
    name: "{{apache_package_name}}"

- name: Start service and enable start on boot
  service:
    name: "{{apache_service_name}}"
    state: started
    enabled: yes

- name: Upload minimal homepage
  template:
    src: index.html.j2
    dest: "{{apache_document_root}}/index.html"
    mode: 0644

- name: Upload config file
  copy:
    dest: "{{apache_config_directory}}/redirect.conf"
    content: |
      Redirect /go https://www.google.com
  notify: reload apache
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
- name: Activate config (only on Debian-like systems)
  command:
    cmd: a2enconf redirect
    creates: /etc/apache2/conf-enabled/redirect.conf
  when: ansible_os_family == "Debian"
```

Listing 8.22 "apache/tasks/main.yml": Role Tasks with External Template File

Next, you transfer the OS-specific variable files as they are into the *vars/* folder of the role, too. However, you should omit the now unnecessary `apache_` prefix in the filenames, as mentioned before:

```
---
apache_package_name: apache2
apache_service_name: apache2
apache_document_root: /var/www/html
apache_config_directory: /etc/apache2/conf-available
```

Listing 8.23 "apache/vars/debian.yml"

```
---
apache_package_name: httpd
apache_service_name: httpd
apache_document_root: /var/www/html
apache_config_directory: /etc/httpd/conf.d
```

Listing 8.24 "apache/vars/rocky.yml"

```
---
apache_package_name: apache2
apache_service_name: apache2
apache_document_root: /srv/www/htdocs
apache_config_directory: /etc/apache2/conf.d
```

Listing 8.25 "apache/vars/opensuse-leap.yml"

```
---
apache_package_name: apache2
apache_service_name: apache2
apache_document_root: /var/www/html
apache_config_directory: /etc/apache2/conf-available
```

Listing 8.26 "apache/vars/ubuntu.yml"

Also, you should place the external *index.html.j2* template file in the *templates/* role's folder. However, there is a small difference between this and the previous version: you

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

no longer provide a default for the `apache_color` variable in the template. This is now moved into the *defaults/* role's folder, which we will discuss below:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>{{inventory_hostname}}: Test</title>
  </head>
  <body bgcolor="{{apache_color}}">
    <h1>Welcome to {{inventory_hostname}}!</h1>
    <h2>Our team:</h2>
    <ul>
    {% for host in groups['all'] %}
      <li>
      {% if host != inventory_hostname %}
      <a href="http://{{ hostvars[host]['ansible_all_ipv4_addresses'] |
                         map('regex_search','^192\..*')              |
                         select('string')                           |
                         first }}">
        {{host}}
      </a>
      {% else %}
        <span style="background-color: LightCyan">{{host}}</span>
      {% endif %}
      </li>
    {% endfor %}
    </ul>
  </body>
</html>
```

Listing 8.27 "apache/templates/index.html.j2": Jinja Template File in "apache" Role

You must not forget the only handler; it belongs (as you probably guessed) in the *handlers/main.yml* file:

```
---
- name: Reload Apache service
  service:
    name: "{{apache_service_name}}"
    state: reloaded
  listen: reload apache
```

Listing 8.28 "apache/handlers/main.yml": Handler File in "apache" Role

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Last but not least, you should ensure a default parameterization of the role. A best practice when creating roles is this: whenever possible, you should provide meaningful default values for all parameters. The role's *defaults/main.yml* file is intended to hold these values:

```
---
apache_color: LightBlue
```

Listing 8.29 "apache/defaults/main.yml"

By the way, role defaults rank second in the precedence list and are therefore overridden by any other mechanism of variable declaration.

Tip

You should preface all your role parameters with a common prefix (which should be `apache_` here) as much as possible. This increases clarity and prevents potential naming collisions.

Don't try the new role yet! First, you should remove any remnants at the following locations that are related to our previous Apache work:

- *playbooks/vars*
- *playbooks/templates*
- *group_vars/all.yml*

Otherwise, something may work by chance during testing because a play will use these locations instead of the role subdirectories, as it should!

You can now test ad hoc or with this playbook that only calls our new role:

```
---
- hosts: all

  roles:
    - name: apache
```

Listing 8.30 "apache4.yml": Playbook to Test "apache" role

If everything runs as expected, our Apache servers should now greet us with a homepage in light blue!

8.5 Documentation (and Conventions)

We still need to address the tricky topic of documentation. Unfortunately, even you will not remember what your roles do after just two weeks—let alone which parame-

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

ters are possible or required. Now, imagine colleagues who are using your role for the first time and don't want to dive into source code from start to finish....

To tackle this problem, you would typically use a combination of two approaches:

1. Good code documents itself.
2. An attached README file takes care of the rest.

The following sections will explain this in more detail.

8.5.1 "defaults/main.yml" as a Convention

Let's generously assume that everyone involved has an idea of what a role named `apache` might be about. Then, these two important questions remain:

- What parameters influence the behavior of the role?
- What are the mandatory parameters, without which the role cannot work at all?

As a coding convention, a team could agree that *all* parameters are always listed in *defaults/main.yml*, along with a short description. Here, you'll see an example that *does not* reflect the functionality of our current `apache` role:

```
---
# Default background color:
apache_color: LightBlue

# Redirect to Google:
apache_google_redirect: /go

# User/password for access to protected areas:
apache_admin_user: boss
apache_admin_pass: # no default
```

Listing 8.31 "defaults/main.yml": Example Commented Role Defaults

Take note of the last line: the intent is to communicate this parameter to the user, but you intentionally leave it unset because using default passwords is never a good idea. From a purely YAML perspective, you could also use any of the methods described in Chapter 4, Section 4.7; it's best for the team to agree on one method.

As a result, the role should exit as quickly as possible if the parameter is not provided by the user. Do you remember the `assert` module? If possible, use it in an early task:

```
---
- assert:
    that:
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    - apache_admin_pass != None
  fail_msg: Please set password with apache_admin_pass
```

Listing 8.32 "tasks/main.yml"

8.5.2 "README.md"

The second approach, which excellently complements certain coding conventions, is to include a README file. A simple text format would be sufficient, but in recent years the *Markdown* format has become the de facto standard for such purposes. Markdown is a simple markup language and quite easy to learn (for example, see *https://www.markdownguide.org/*). Moreover, Markdown has the advantage that it's easily readable with a simple text viewer (unlike formats such as HTML or LaTeX).

If you need to, you can easily convert Markdown into "nicer" formats; Git servers with a graphical interface like Gitea or GitLab do this automatically. Ansible Galaxy does it, too, and the `ansible-galaxy init` command produces a *README.md* that exactly matches the specifications there.

If you don't plan to publish your roles on Galaxy, you should develop a template that your company approves and that is well-received by everyone involved. An example is the template in Listing 8.33:

```
# Ansible Role: apache

A simple Apache installation on various Linux systems

## Prerequisites
You must disable the firewall or open port 80.

## Suitable for
Debian 12, Rocky 9, openSUSE 15, Ubuntu 24.04

## Parameters with Default Values

    apache_color: LightBlue
    apache_google_redirect: /go
    apache_admin_user: boss
    apache_admin_pass: # no default

## Description
This role was created purely for demonstration purposes. It installs
an Apache web server on various Linux systems, starts it,
and sets up a simple homepage. A redirection rule
is also configured according to Google.
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
You can determine the background color of the homepage with the `apache_color`
parameter. All other parameters are for internal use only.

## Example Playbook

```
- hosts: all

  roles:
    - name: apache
```

Author
John Doe <john@example.org>
```

Listing 8.33 "apache/README.md": Example Role Documentation

You can edit this with any text editor, of course. However, if you want to work as you write, we recommend that you to use an editor like *ReText*, which offers a preview or WYSIWYG mode.

Man Pages

With the *pandoc* software, you can also generate a classic man page from the Markdown format. Try the following command as a basis for your own ideas:

```
$ pandoc -s roles/apache/README.md -t man | man -l -
```

8.6 Reusing Roles

Even after everything you've learned in this chapter so far, you may still be wondering how to reuse roles in practice or use them across multiple projects. We have placed roles in the *roles/* directory of our Ansible project, making them available to all playbooks within this project—but what about the next project that also needs to use these roles?

A logical and effective approach would be to move the roles into their own independent project. You should also organize them into a collection structure, which requires Ansible 2.10 or a more recent version. (While you could set up something similar with older versions of Ansible and without collections, collections are standard in Ansible. So, if you need to, you can upgrade from your outdated version and embrace the modern approach!)

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

8.6.1 Extending the Directory Structure

To demonstrate what this could look like, we'll continue with a copy of the previous project. Everything worked fine there, and we definitely don't want to lose that state:

```
$ cd ~/ansible/projects
$ cp -a start demo1
```

To make your start with collections, you create your own directory structure and switch into it:

```
$ mkdir -p ~/ansible/collections/ansible_collections && cd $_
```

Important

Unfortunately, the redundant *ansible_collections* intermediate directory is technically necessary!

Then, you create the basic layout for a new collection as well as a folder for roles:

```
$ ansible-galaxy collection init local.www
- Collection local.www was created successfully
```

The roles folder should already exist, but better safe than sorry:

```
$ mkdir -p local/www/roles
```

Now, the Apache role can "move" from its previous location into the collection:

```
$ mv ../../projects/demo1/roles/apache local/www/roles
```

You can use the FQCN in a project's playbook that wants to use this role. For example, you can do it like this:

```
---
- hosts: all

  roles:
    - name: local.www.apache
```

Listing 8.34 "apache5.yml": Test Playbook for Collection Role

You haven't found the role yet because we've "installed" its parent collection in a non-standard location.

Let's imagine for a moment that both our project and the collection are available as a Git repository, as Table 8.1 describes:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Directories	Git Repositories
~/ansible/projects/demo1	`ansible/project-demo1`
~/ansible/collections/ansible_collections/local/www	`ansible/collection-local-www`

Table 8.1 Hypothetical Git Repositories

(If you have a Git server that's comparable to the one from Chapter 9, you're welcome to implement these repositories for testing purposes. Otherwise, read on for now, believe it for the time being, and try it out later if necessary.)

Our goal now is that a hypothetical user will only need to check out the `project-demo1` project and then, in step 2, reinstall all required collections through a simple automation. Additionally, users should ideally not need to worry about these internal dependencies. It's also important to consider not only human users but also web applications, like the ones in Chapter 9, which may eventually need to trigger your playbooks. In such cases, the issue becomes even more apparent, as an application can't read a *README.txt* file that says, "You must install these collections beforehand!"

The solution for both humans and machines is to use a *collections/requirements.yml* project file, where all dependencies are specified in a standardized format:

```
---
collections:
  - name: http://gitea.example.org/ansible/collection-local-www.git
    type: git
```

Listing 8.35 "collections/requirements.yml": Defining Dependencies on External Collections

With the following command, you can now install all listed dependencies, which will then end up in the default collection search path *~/.ansible/collections/*:

```
$ ansible-galaxy install -r collections/requirements.yml -f
```

The `-r` option expects the requirements file as a parameter, and `-f` forces overwriting of already installed collections to ensure you get the latest version from the version control system.

Important

Since you have to specify the name of the requirements file anyway, you could have named it anything and located it anywhere. But web applications like AWX expect and process the *collections/requirements.yml* file by default.

You should adhere to this rule—unless you are absolutely sure you will not use such tools in the future.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

8.6.2 The Developer's Situation

If you find yourself in a situation where you're developing both collections and related projects, then both directories are already in the latest version on your hard drive, and you probably won't feel like running the above `ansible-galaxy install` command after every small change in the collection (not to mention the necessary preceding `git commit/push`.)

Instead, you could extend the collection search path as described in Chapter 7, Section 7.1.2. It's most advisable here to set the `ANSIBLE_COLLECTIONS_PATH` environment variable at an appropriate place (e.g., *.bashrc*, *.envrc*):

```
export ANSIBLE_COLLECTIONS_PATH=~/ansible/collections
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 9
Web Interfaces: AWX and More

In this chapter, we'll focus on web applications that are related to Ansible. I'll introduce you to some very different applications—from a small, lean tool that produces an informative website to an all-in-one solution for everything that requires several gigabytes of memory.

Important

This section will enable you to do your own evaluation of the applications in question, and it'll do it by describing the quickest and simplest way possible to set them up in our lab environment.

However, it's not my intention to give you detailed setups of production quality (possibly with high availability and backup strategies)! You'll have to assess which solution is appropriate for you.

9.1 Installation of Python Packages on Current Debian and Ubuntu Systems

In this section (and also later), you'll occasionally need to install Python packages using the Python package manager. Therefore, you should first ensure that the *python3-pip* distribution package is installed on your system.

You may have already noticed that since Debian version 12 and Ubuntu version 23.04, attempting to install a module with `pip3` results in an error message, as follows:

```
# (Just an example; we do not need this module.)
# pip3 install matplotlib
error: externally-managed-environment
This environment is externally managed

    To install Python packages systemwide, try apt install
    python3-xyz, where xyz is the package you are trying to
    install.

    If you wish to install a non-Debian-packaged Python package,
    you should create a virtual environment using python3 -m venv path/to/venv.
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    Then, use path/to/venv/bin/python and path/to/venv/bin/pip. Make
    sure you have python3-full installed.

    If you wish to install a non-Debian-packaged Python application,
    it may be easiest to use pipx install xyz, which will manage a
    virtual environment for you. Make sure you have pipx installed.

    Note: If you believe this is a mistake, please contact your Python
    installation or OS distribution provider. You can override this,
    at the risk of breaking your Python installation or OS, by
    passing --break-system-packages. Hint: See PEP 668 for the
    detailed specification.
```

Of course, in this case, it would be better to use the *python3-matplotlib* distribution package. But in our cases, there is usually no adequate distribution package.

Environments with virtualenv are indeed nice, but because you have to activate them first, they can be a bit annoying. And when there is a proliferation of such environments, you can lose track of them (along with your composure). And alas, the suggested *pipx* only helps in some cases.

At that point, the only option left is `--break-system-packages`. It sounds bad, but it simply means "Just do it—in the last fifteen years, it's almost always worked out fine!" At least for our lab, it's the simplest solution.

To avoid having to specify this option with every `pip3` call, I will assume from now on that this setting is set in the */etc/pip.conf* global configuration on the corresponding Debian-like lab system:

```
[global]
break-system-packages = true
```

Listing 9.1 "/etc/pip.conf"

You can always check this setting with `pip3 config debug`.

9.2 Ansible Configuration Management Database ("ansible-cmdb")

With the small Python-based `ansible-cmdb` tool (see *https://github.com/fboender/ansible-cmdb*), you can look at the Ansible facts on a cleanly designed HTML.

Installing this tool on the control host is quick and easy with the Python package manager:

```
$ sudo pip3 install ansible-cmdb
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Unfortunately, the not-so-up-to-date software is somewhat inflexible regarding the detection of the Python interpreter. The program insists on using an executable named `python`, which can be a problem on current systems if only `python3` is directly callable:

```
$ ansible-cmdb
No suitable python version found (v2.7 or higher required). Aborting
```

To avoid further system interventions in this case, I suggest using a quick fix in which you define an alias that calls the actual `ansible-cmdb.py` program by using the “correct” Python interpreter:

```
$ alias ansible-cmdb="python3 /usr/local/lib/ansiblecmdb/ansible-cmdb.py"
```

9.2.1 Usage

If you’re already operating a facts cache with JSON files (see Chapter 6, Section 6.3.2), you can use it immediately:

```
$ mkdir -p /tmp/cmdb
$ ansible-cmdb -f ~/.ansible/fact_cache > /tmp/cmdb/index.html
```

With `-i`, you can also specify an inventory explicitly, and the output file location will be completely up to you.

If you have not yet set up a facts cache, then you should gather fresh facts and write them to a directory so you can generate the report page from it:

```
$ mkdir -p /tmp/cmdb
$ ansible all -m setup --tree /tmp/facts.d
$ ansible-cmdb /tmp/facts.d > /tmp/cmdb/index.html
```

9.2.2 Deploying the Page

You can now view the generated report page in a browser of your choice. However, on a nongraphical system, this is not quite so simple. You could, for example, start a temporary web server with Python and serve the page to the outside this way:

```
$ python3 -m http.server -d /tmp/cmdb
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

At a URL like *192.168.150.100:8000*, you can now admire content comparable to that in Figure 9.1.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

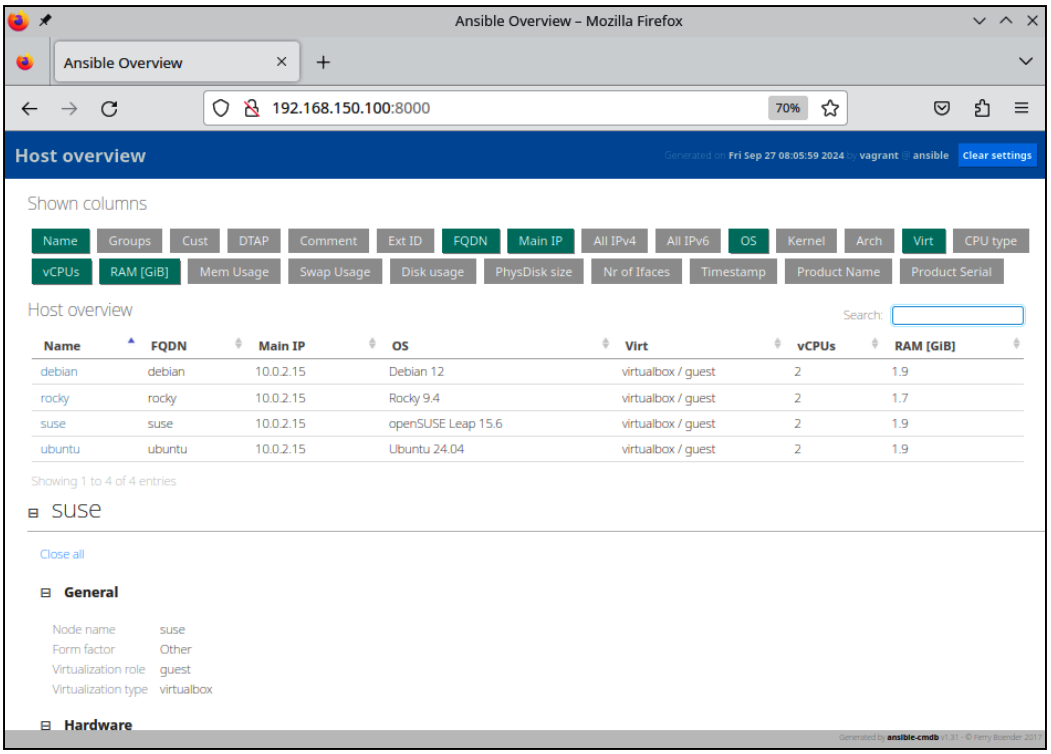


Figure 9.1 ansible-cmdb

9.3 Preparations for Operating More Demanding Applications

We will make extensive use of the Kubernetes software in the specific form of *K3s*, as almost all of the web applications to be considered are provided as container images. The reason we are using Kubernetes instead of a simple Docker environment is that AWX, as the most important application, is now officially only supported on Kubernetes.

If this doesn't work with containers or even Kubernetes, don't worry. It's just a means to an end for us, and you can largely ignore the technology behind the scenes.

The preparations you need to make for operation are as follows:

1. Determine which of your systems should become the K3s server.
   For simplicity's sake, let's assume that our control host should also take on this task.
2. Ensure that there is enough main memory available.
   If you want to test Ansible AWX later, you should have at least 4 GB of RAM; with 5 to 6 GB, you would be on the very safe side if other applications were added.

   Do you still remember your *Vagrantfile* from Chapter 1, Section 1.4? If so, you should simply allocate the desired main memory to the `ansible` machine, like this:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
[...]
  config.vm.define "ansible" do |ansible|
    # [...]

    # Please add the following three lines:
    ansible.vm.provider "virtualbox" do |v|
      v.memory = "4096"
    end

  end
[...]
```

Listing 9.2: "Vagrantfile": Example Excerpt for Increasing Main Memory of VM

To activate the new setting, you simply execute this:

```
$ vagrant reload ansible
```

A moment later, the virtual machine will be back online, and you should log in again and test with `free -h` to see whether the desired amount of main memory has been allocated.

3. Install Ansible and Git.
   Of course, an Ansible community package is already available on our control host, but I'm mentioning it here again for completeness:

   ```
   $ sudo apt install ansible git
   ```

4. Clone the `ansible-buch/cbi` GitHub repository.
   You should clone it into a new directory area (e.g., *~/book*) and then switch to the cloned project folder:

   ```
   $ mkdir -p ~/book; cd $_
   $ git clone https://github.com/ansible-buch/cbi.git
   $ cd cbi
   ```

   CBI stands for *container-based infrastructure*. It's a small framework that I developed, and it allows you to manage containerized applications in test environments or smaller private setups in a relatively simple way.

5. Create a configuration file.
   In the project folder, you'll find the `cbi` program that offers various subcommands. You should use the `config` subcommand:

   ```
   $ ./cbi config
   ```

   This creates the *env.yml* file inside the project folder. You're welcome to look at or modify it, but in all subsequent call examples, I'll assume that everything has been left in its original state.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

The file defines a default password for all applications. At the time of writing this book, it's this:

```
Secret-1234
```

However, it may change in the future due to increasing complexity requirements. Therefore, I will not constantly repeat it in the following sections, but I'll always use `<The password from env.yml>` in the appropriate places.

6. Install the Kubernetes ecosystem.
   You need to provide the setup routine with two important pieces of information by answering the following questions:
   - Which interface will provide access to your system from the outside? If in doubt, you can use the `ip -br a` command to list your interfaces. You can then set the `iface` variable to the interface you select when calling **`cbi`**.
   - Should an unprivileged account be allowed to execute Kubernetes commands in addition to root? If so, you should set the `user` variable to that user ID on the command line, too.

   In our lab, the setup call will probably look like this:

   ```
   $ sudo ./cbi up k3s/environment -e iface=eth1 -e user=vagrant
   ```

7. Check whether the K3s server is operational.

   ```
   $ kubectl get nodes
   NAME      STATUS   ROLES                  AGE   VERSION
   ansible   Ready    control-plane,master   27s   v1.30.5+k3s1
   ```

8. Check whether name resolution *in a container* works correctly.
   If name resolution does not work immediately, wait a bit and then repeat the `nslookup` command. The internal name service also needs a little time:

   ```
   $ kubectl run a1 -it --rm --image=alpine -- sh
   / # nslookup ansible.com
   […]

   Anything from the domain configured in env.yml must be resolvable:
   / # nslookup foo.example.org
   […]

   Leave pod:
   / # exit
   ```

9. Ensure appropriate name resolutions outside of Kubernetes.
   You need a fully qualified name that can be resolved to the primary IP address of the container host (in our example, this is `192.168.150.100`) for accessing each web application you want to test. The complete list of all names looks like this:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
gitea.example.org
awx.example.org
ara.example.org
semaphore.example.org
```

Let's work with classic `/etc/hosts` entries, and I want to emphasize again that you should do this *on the control host and on the system where your browser is running*. The possibly long line with all names will then look like the following (and as mentioned, you can simply add the names of all applications you want to test):

```
192.168.150.100  gitea.example.org awx.example.org ara.example.org […]
```

Listing 9.3: "/etc/hosts": Excerpt

(You will probably at least need Gitea and AWX; otherwise, you wouldn't be reading this chapter.)

If your base system is Windows, you'll need to edit the *%windir%\system32\drivers\etc\hosts* file with administrator rights. The syntax is identical to that of the Linux variant.

Important

The name resolutions are crucial and indispensable for the test environment! All "containerized" web applications will be behind a reverse proxy, and the test environment will only be able to route traffic by the name of the respective application.

So, pay meticulous attention to the correct spelling. If you make a typo, you will not be able to see the application in question in the lab!

9.4 The Gitea Git Server

Typically, all web frontends for Ansible expect that Ansible projects will be managed in a version control system—and nowadays, that is Git 99% of the time. So, we definitely need our own Git server.

The application *Gitea* will become our key Git server. The software is very lightweight yet feature rich, and its GitHub-inspired web interface makes it a pleasure to use. All in all, it is widely adopted in production environments.

9.4.1 Commissioning and Logging In for the First Time

1. Start the Gitea server:

```
$ cd ~/book/cbi

$ sudo ./cbi up k3s/gitea
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

The process can take some time, so please be patient here. (The only reason why we need `sudo` is because the setup procedure places a global git config file in */etc* as a last step.)

2. Log in to the web interface.
   To do this, you open the *https://gitea.example.org* page in your browser. You'll need to accept the self-signed certificate by using the usual procedure in your browser, and once you get to the homepage, you should click on **Login** at the top right.

The login credentials are as follows:

```
Username: root
Password:     <The password from env.yml>
```

9.4.2 Checking In on Our Initial Project

Let's use version control to commit our very first Ansible project and push it to the new Git server. You should proceed step by step as follows:

1. Create a new organization in Gitea.
   To do this, you click on **+** at the top right in the web interface and then click on the **New Organization** subitem. You should use the appropriate name `ansible`, and you can leave all other settings as the defaults.
2. Create a new repository in Gitea.
   For this, you click on **+** at the top right in the web interface then click on the **New Repository** sub-item. You should also do the following:
   - Make sure that `ansible` is selected as the owner!
   - Choose a repository name that clearly conveys its purpose; a reasonable choice would be `project-start`. All other settings can stay at their defaults.
3. Return to the command line. You'll now continue as an unprivileged Ansible user in our project:

   ```
   # As an unprivileged Ansible user!
   $ cd ~/ansible/projects/start
   ```

4. Add IP addresses to the inventory.
   Soon, for the first time, it will no longer be a user from the control host that will start the playbooks—instead, it will be an application in a Kubernetes container. However, such an application will not be able to understand names like `suse` or `ubuntu`.

   You can fix this easily by expanding the entries in your static inventory with address information, like this:

   ```
   [test_hosts]
   debian ansible_host=192.168.150.10
   rocky  ansible_host=192.168.150.20
   suse   ansible_host=192.168.150.30
   ```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
ubuntu ansible_host=192.168.150.40

# ...
```

Listing 9.4: "inventory": Inventory with IP Addresses

5. Commit your project into version control:
   Take the opportunity to tidy up a bit first (with *.retry* files or similar ones). If the project folder looks clean enough, then you can enter the following commands:

   ```
   $ git init
   $ git add .
   $ git commit -m "Initial commit"
   ```

   You can't break anything with this; the *.git/* subfolder will be created as storage of additional information.

   (Here's another note for Git experts: During the Gitea deployment, CBI also placed a global Gitconfig in */etc/gitconfig*. That's why the commit works "just like that" here.)
6. "Push" the whole thing to your new Git server, as follows:

```
$ git remote add origin https://gitea.example.org/ansible/project-start.git
$ git push -u origin main
Username for 'https://git.example.org': root
Password for 'https://user1@git.example.org': <The password from env.yml>
[...]
```

For future pushes, a simple `git push` will suffice.

Figure 9.2 shows how the project page should look after a browser reload.

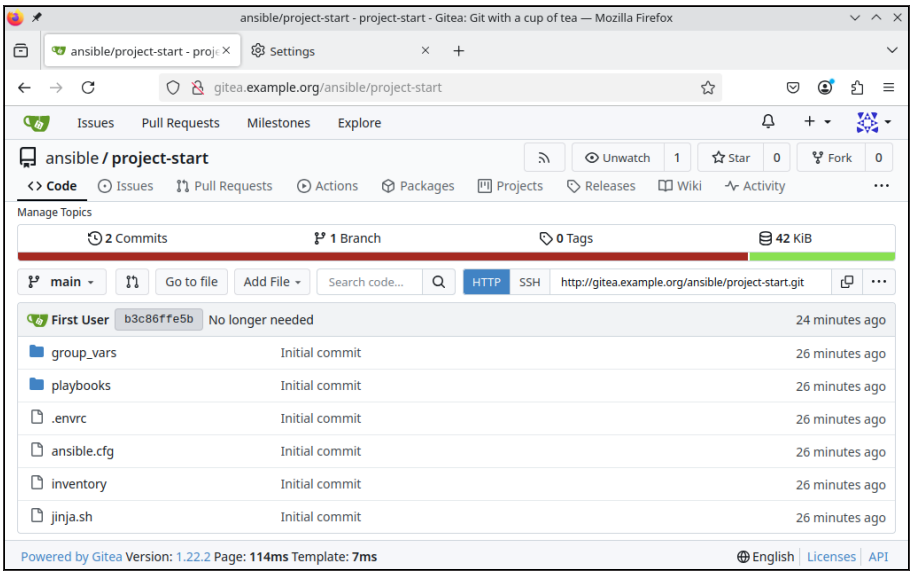


Figure 9.2 Gitea Page with Project View

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

9.4.3 Adding »README.md« and Useful Git Commands

Just as with roles, it's a good idea to add a *README.md* file to projects in the base folder. A bit of documentation is always useful, and furthermore, many graphical tools will format and present such a file nicely by default.

So, for example, you can place something like this in the root directory of your project:

```
# Our First Ansible Project

## Inventory

The project requires four virtual machines, as listed in the `inventory` static
inventory file.

## Playbooks

In the `playbooks` folder, you will find various playbooks. You can start them
with `ansible-playbook`; like this:

```
ansible-playbook playbooks/hello-ansible.yml
```

Roles

- The `apache` role sets up an Apache web server on four different
   distributions.
- The `hallo` role is a first, very simple example of a role.
```

Listing 9.2 "README.md": Example Greeting File

Add this file to the repository via the command line:

```
$ git add README.md
$ git commit -m "Added README.md"
$ git push
```

And since we are on the topic of Git, here's an overview of the most important Git commands that will help you "survive" on the command line for quite a while:

```
You use these to track changes:
$ git status
$ git log  [--stat] [<FILES...>]
$ git diff          [<FILES...>]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
You use these to check on changes or new files:
$ git add       [<FILES...>]
$ git commit -m "<COMMENT>"
$ git push
```

Note

Git can do much more, of course—after all, Linus Torvalds invented it to manage the development process of the entire Linux kernel. If you want to learn more about Git, take a look at *Git: Project Management for Developers and DevOps Teams* by Bernd Öggl and Michael Kofler (Rheinwerk Computing, 2023, *www.rheinwerk-computing.com/5555*).

9.5 AWX

AWX (*https://github.com/ansible/awx*) is essentially the in-house web interface for Ansible.

Red Hat, Inc. uses this free software as the basis for its *Ansible automation controller* commercial product (*https://www.ansible.com/products/controller*), which was formerly better known as *Ansible Tower*.

The advantages of Ansible AWX over the Ansible command line are as follows:

- Custom user management with access rights to inventories, playbooks, and more.
- Reporting/traceability.
- Workflows: chaining of playbooks.
- REST API: Everything that can be done via the interface can also be automated through API calls.

And the disadvantages of Ansible AWS relative to the Ansible command line are as follows:

- Officially, only a container-based setup in Kubernetes is supported, and that may not be everyone's preference.
- It is quite large, so you need to allocate a couple of gigabytes of memory to it.

9.5.1 Commissioning and First Login

This is how you quickly set up a test environment:

1. Use the cloned GitHub repository here again:

```
$ cd ~/book/cbi
$ ./cbi up k3s/awx
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

2. Once all the associated containers are running, you can open the application at *https://awx.example.org*. However, on the first start, it may take *a few* minutes.

Once the `cbi` command has completed, you can observe its doings with `kubectl get pods -n awx`. As soon as all pods involved have reached a status of `Running` or `Completed`, the web interface will be usable. (You can access the website before it's ready, but you'll only see a status message from AWX.)

The login credentials are as follows:

```
Username: admin
Password: <The password from env.yml>
```

Figure 9.3 shows the application's starting page.

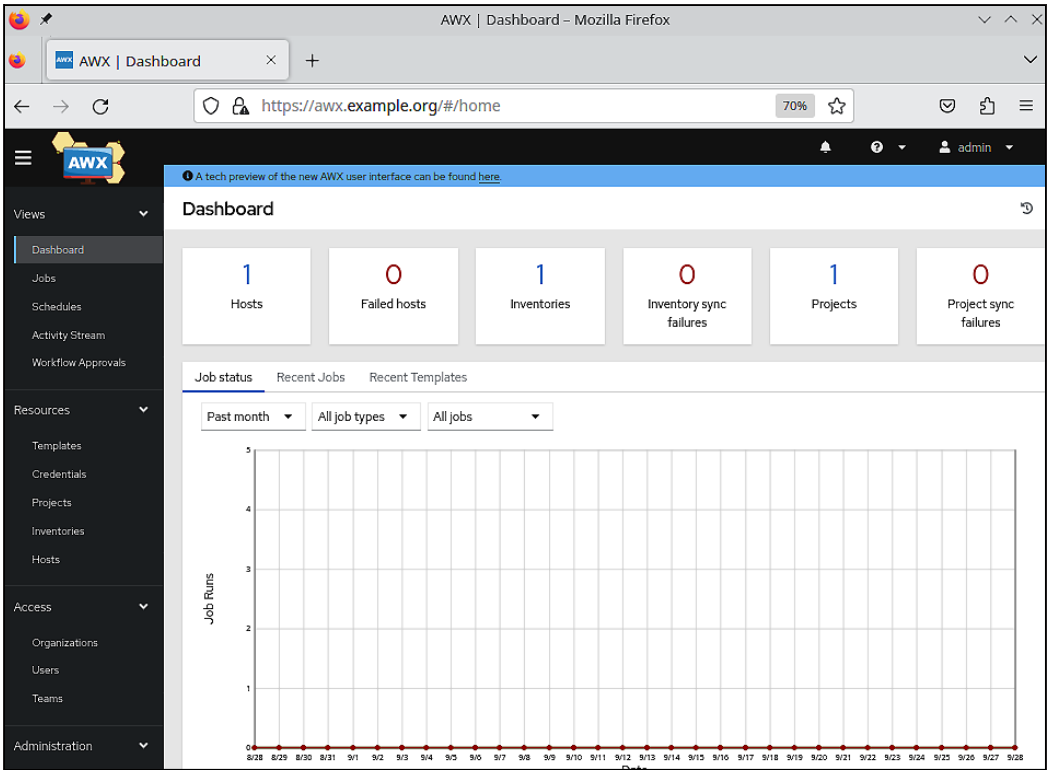


Figure 9.3 AWX Starting Page

9.5.2 Example Usage

As an example, we will now go through the process of how to create a new project in AWX, with the final goal of starting a playbook with it. As already mentioned, AWX is a powerful tool that offers many possibilities—but you need solid knowledge of Ansible to get the most benefit from it.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Before we get started, take note that you should always make sure to use the blue **Add** button to create new objects!

Now, you'll create a project and start a job by following these steps in AWX:

1. Add the private SSH key for your Target Hosts under **Credentials**:
   - Name: `SSHkey1` (This is freely selectable.)
   - Organization: `Default`
   - Credential Type: `Machine`
   - SSH Private Key: Copy the contents of your *~/.ssh/id_rsa* here.
2. Create a new project under **Projects**:
   - Name: `Project1` (This is freely selectable.)
   - Organization: `Default`
   - Source Control Type: `Git`
   - Source Control URL: `http://gitea.example.org/ansible/project-start.git`

   Please do not use *HTTPS* here, as AWX does not like self-signed certificates without further configuration.

   Also, feel free to check **Update Revision on Launch** (if you want to automatically have the latest version from the Git repository with each run).

   Make sure that after saving, the automatic synchronization is successful!
3. Add a (normal) inventory in **Inventories**.

   First, you just provide a name like `Inv1` and click on **Save**.

   Next, you can add a source in the **Sources** tab (for example, `InvSource1` with the `Sourced from a project` setting). Then, you can select your appropriate project and a suitable inventory file, and you must save the source.

   With the **Sync** button, you can then start the synchronization process for your source. (This can take quite a long time the first time).

   Hopefully, the synchronization will be successful, and afterward, the corresponding targets or groups should become visible under **Hosts** and **Groups** in the inventory.
4. Finally, you create a job template in **Templates**.

Fill in all the necessary fields. Do not forget to select the SSH key under **Credentials**!

Once you've provided all the information, save it and then click on **Launch**!

9.5.3 Execution Environments

Maybe you've already noticed the term *execution environment* (EE) during your initial attempts with AWX. This term refers to the clever idea of packaging all the tools, collections, and other dependencies you need to execute Ansible playbooks into a container

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

image. By doing this, you achieve maximum flexibility—especially because you can even build your own EEs.

If you do not select a specific EE in AWX, the default EE will be `quay.io/ansible/awx-ee`. If you're curious, you can start a pod in Kubernetes and take a look at it "from the inside":

```
# This is what we use in our case as root:
$ kubectl run ee1 -it --rm --image=quay.io/ansible/awx-ee -- bash

bash-5.1$ ansible --version
ansible [core 2.15.12]
bash-5.1$ ansible-galaxy collection list
[…]

bash-5.1$ exit
```

As you've seen, while some collections are on board, the *community.general* collection is not (as of September 2024). This would be painfully noticeable if you ran one of our Apache playbooks with AWX involving a SUSE system. The `zypper` module we need for package management belongs to this very collection, so that gives us a good reason to build our own execution environment!

Building an Execution Environment

If you're familiar with containers, you can build an EE image in the traditional way with a Dockerfile. However, it is somewhat more convenient to use the *ansible-builder* program since we installed it using Pip. We also recommended that you create your own directory for building, as follows:

```
$ sudo pip3 install ansible-builder

$ mkdir -p ~/build-ee && cd $_
```

Next, you need a definition file. You should stick to the special *execution-environment.yml* name because the file will then be automatically found by `ansible-builder`:

```
---
version: 3

dependencies:
  ansible_core:
    package_pip: ansible-core==2.14.16
  ansible_runner:
    package_pip: ansible-runner
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
  system:
    - openssh-clients

  galaxy:
    collections:
      - name: awx.awx
      - name: ansible.posix
      - name: community.general

images:
  base_image:
    name: quay.io/centos/centos:stream9
```

Listing 9.3 "execution-environment.yml": Minimal Definition File for Execution Environment

I chose an *ansible-core* with version 2.14.16 because it has already proven itself in our working environment. The file format is very readable, and you'll find a description at *https://ansible.readthedocs.io/projects/builder/en/latest/definition/* with a list of all possibilities.

The `ansible-builder` program can use Docker or Podman. In this case, I'm using Podman internally because my CBI setup procedure has already installed it. So, let's start the build process. You should pay close attention to the correct image name:

```
$ ansible-builder build -vvv -t gitea.example.org/ansible/my-ee
(Have some patience.)
```

What does all this have to do with Gitea? We need to upload the (hopefully) successfully created image to a container registry so that AWX can download it from there. (That's how it works in the container world....) Thankfully, our Gitea server also offers registry functionality, and in the container universe, the name of the repository is baked into the image name. After the build process, you should do a little cleanup:

```
$ podman image prune -f
```

Next, you log in to Gitea via the registry access with the usual Gitea credentials, and then, you can push the container:

```
$ podman login gitea.example.org
Username: root
Password: <The password from env.yml>
Login Succeeded!

$ podman push gitea.example.org/ansible/my-ee
Getting image source signatures
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
Copying blob 003a46ce89c2 done
Copying blob f99e5b4ec4e8 done
[…]
```

Note

Container images are hard to find in the Gitea WebUI. If you're interested, you can switch to the main page (the green teacup), click on **Organization,** and select `ansible` under **My Organizations**. You will find them under the **Packages** tab.

Integrating the new Execution Environment into AWX

The rest is almost child's play. You just need to integrate our new EE into AWX, as follows:

1. Add a new EE under **Execution Environments**.
   - Name: `my-ee` (which is freely selectable)
   - Image: `gitea.example.org/ansible/my-ee:latest`

     Important: You must specify the tag, even with `latest`!
   - Pull: `Always pull container before running.`

     The `Always` pull option does have some overhead, but it ensures that you always use the latest version of the EE image. If you consistently work with version numbers instead of `latest`, you can of course also use the more efficient `Only pull if not present`.
2. Edit your job template under **Templates**. You should select your new execution environment from the dropdown list.

Don't forget to save! A relaunch should hopefully proceed as desired now.

9.5.4 Conclusion

As you may have already noticed, in terms of functionality, nothing compares to AWX. At first glance, many aspects of it seem complex, but once you dive in, everything is ultimately logical and clear.

Why even consider other applications at all? The only reasons I see are the high resource requirements and the necessity of a Kubernetes environment. (And if you're new to Kubernetes, please do not compare our simple lab setup to professionally configured Kubernetes clusters!)

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

9.6 ARA

ARA (*https://github.com/ansible-community/ara/*) is an application for post-analysis of playbook runs. It is not in direct competition with AWX, and it could thus become another pillar in your Ansible environment.

ARA collects information during playbook runs, which end up in a database and can later be evaluated in a web frontend or the command line.

9.6.1 Test Setup

Since ARA is quite closely integrated with Ansible through plug-ins, containers are not necessary here. You'll install the Python package locally, which you could in principle even do unprivileged. However, a systemwide installation like this will simplify the calls later:

```
$ sudo pip3 install "ara[server]"
```

Next, you connect Ansible to ARA by using a callback plug-in:

```
$ export ANSIBLE_CALLBACK_PLUGINS="$(python3 -m ara.setup.callback_plugins)"
```

(If you want to use ARA permanently, you can also work with the `callback_plugins` directive in the *ansible.cfg*.)

Then, you need to determine the domain names that are allowed to access the ARA server:

```
$ export ARA_ALLOWED_HOSTS='["127.0.0.1", "localhost", "ara.example.org"]'
```

Now, you start any playbooks. Upon its first start, you'll notice that ARA initializes its *~/.ara* data area:

```
$ ansible-playbook <ANY_PLAYBOOK.yml>
[ara] No setting found for SECRET_KEY. Generating a random key...
[ara] Creating data & configuration directory: /home/vagrant/.ara/server
[ara] Writing default settings to /home/vagrant/.ara/server/settings.yaml
Operations to perform:
  Apply all migrations: admin, api, auth, contenttypes, db, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
[…]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

You now have the `ara` command line tool available for analysis:

```
$ ara playbook list
[…]
```

And you also have much more; see `ara --help`.

Alternatively, you can start the built-in web interface on a suitable free port of your control host:

```
$ ara-manage runserver 0.0.0.0:8080
```

You can now access this at *http://ara.example.org:8080* (assuming proper name resolution).

When you've finished your tests, you should simply terminate the web server process by pressing Ctrl+C. You should also end data collection by disabling the callback plug-in:

```
$ unset ANSIBLE_CALLBACK_PLUGINS
```

9.6.2 Further Possibilities

If you want to explore the tool in more detail, your first point of reference should be the comprehensive documentation at *https://ara.readthedocs.io/en/latest/*. There you will find information on the following things:

- Various installation options
- Server operation of ARA
- Integration of "real" databases like PostgreSQL
- Operation with container images
- The *ara*-API client

9.7 Other Applications Not Considered in Detail Here

To wrap up, I would like to briefly discuss some other applications.

Important

If AWX is still in operation, it might now be running tight with 4 GB of RAM. So, if you want to start more apps, you would either have to increase the main memory again or stop AWX (Section 9.8).

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

9.7.1 Semaphore

Semaphore (*https://semaphoreui.com*) is an older project that has recently gained some momentum again. Compared to AWX, it is rather lightweight and offers significantly fewer features. The fact that it has no dependency on Kubernetes is certainly an advantage for many.

If you like, you can take a quick look at it via CBI:

```
$ cd ~/book/cbi

$ ./cbi up k3s/semaphore
```

Assuming the correct name resolution, you can log in to the browser at *https://semaphore.example.org* after the usual short waiting time. The login credentials are as follows:

```
Username: admin
Password: <The password from env.yml>
```

You'll find that the ideas here are quite similar to those in AWX. You must first set credentials, repositories, and inventories, and then, you can link everything in job templates.

Here's my assessment: if you don't need much more than "press button, playbook runs," and if you shy away from the complexity of AWX, then Semaphore can be a good choice for you. However, the app would benefit from a few more bells and whistles: you have to manually enter paths to playbooks blindly, and you must enter inventory information using copy-and-paste. Many other fine-tuning options, such as repository synchronization details, are also missing. If you can live with that or if you even see a certain minimalism as an advantage, then you should feel free to explore Semaphore a bit more deeply. As an open-source project, it also thrives on participation.

9.7.2 Polemarch

Polemarch (*https://polemarch.org/*) is a Python/Django-based web GUI for Ansible. It appears to be relatively slim and manageable, and it's definitely worth a look. The project seems to be somewhat dormant at the moment, and the user base is probably quite small—at least that's the impression you get when you start googling for solutions to problems.

Unfortunately, my brief attempt to deploy version 3.0.3 (as of October 2024) in Kubernetes failed because a fundamental Django issue related to reverse proxies arises here, and I couldn't easily solve it with the available image (keyword: CSRF Protection).

If you'd like to briefly see the application anyway, you can start a simple container with Podman:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
# podman run -d -u root --name polemarch \
    -e POLEMARCH_DB_TYPE=sqlite \
    -e POLEMARCH_DB_HOST=/data/polemarch.db \
    -p 8080:8080 \
    docker.io/vstconsulting/polemarch
```

Then, after a short wait, you can access the application via the IP of your container host and the chosen port in the browser: *http://192.168.150.100: 8080*. The login credentials are as follows:

```
Username: admin
Password: admin
```

After the tests, you can remove the container again as follows:

```
# podman rm -f polemarch
```

Here's my assessment: Polemarch is similar to Semaphore in that its simplicity can be an advantage. The fact that the application cannot currently run smoothly in typical reverse proxy environments is somewhat annoying, but I will continue to monitor the project.

9.7.3 Jenkins

Jenkins (*https://www.jenkins.io*) is a very popular continuous integration/continuous deployment (CI/CD) system. If you are already using it in software development, an Ansible integration is certainly a logical step and is desirable. You can do this with a plug-in at *https://plugins.jenkins.io/ansible/.*

Here's my assessment: If you're already familiar with Jenkins, you'll appreciate the Ansible integration. However, if Jenkins has not played a role in your IT life so far, you should not change that just because of Ansible.

9.7.4 Rundeck

Rundeck (*https://rundeck.org*) is a quite popular IT automation tool that has certain overlaps with Ansible. You can perform the Ansible integration with a plug-in at *https://github.com/Batix/rundeck-ansible-plugin*.

Here's my assessment: If you're already using Rundeck, it's fine. Otherwise, it can be quite confusing—just as Jenkins can be.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

9.8 Terminating or Deleting Applications That Are No Longer Needed

In the previous sections, you started some very resource-intensive applications—and now, you can terminate applications that you don't need anymore. You have two options available to do this:

- `./cbi stop <app_path>`

  With the `stop` subcommand, you can remove all containers of the application, but the data and metadata will remain intact. If necessary, you can revive the application at any time with a corresponding `start` command.

- `./cbi down <app_path>`

  With the `down` subcommand, you can discard everything related to this application. If you wish to start the application again later, you'll have to start completely from scratch.

Note

Depending on your interest, you might need the Gitea server again for other topics, so you should at least retain its data.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 10
Additional Tools and Techniques

In this chapter, I'll introduce you to some additional tools and techniques—and a bit of livestock. These include the following:

- Ansible Vault
- Debugging and troubleshooting
- Accelerating playbooks with pipelining
- Talking cows
- Ansible in pull-mode

10.1 Ansible Vault

Now, let's tackle a problem we've been avoiding for a long time: we need a strategy for handling sensitive data like passwords, private keys, etc. In Chapter 2, Section 2.6, I asked you to wait; now, we'll address it.

10.1.1 Before All Technology

Anyone who has been in IT long enough will not expect miracles at this point. It always works the same way:

- What do you do with sensitive plaintext information?

  → We encrypt it.
- And how can you work with it, then?

  → We decrypt it.
- And what does that usually mean from a practical perspective?

  → We have to enter a decryption password.
- And what about processes that are supposed to run largely unattended (or just conveniently)?

  → They also receive some form of authorization to decrypt, meaning they get an access token, directly receive the decryption password, or are allowed to ask an agent that holds the required information.
- So, in that case, is there still sensitive information available in plaintext somewhere?

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

→ Yes, but it is not as obvious or is only temporary (as with the agent solution). That's why we define everything now as secure and good.

Does all of that sound familiar to you? Think, for example, of the popular SSH public key method:

- To avoid constantly entering account passwords, you generate a key pair and install the public key on the desired target systems. (This is not just about convenience: passwords often end up on sticky notes stuck to monitors, and the threat of keyloggers is very real.)
- Since anyone who possesses the private key can log in to the corresponding target systems, you need to encrypt the private key. That makes a key theft of the private key no longer *immediately* problematic.
- However, you would need to decrypt the private key for each login process, meaning you'd need to enter a passphrase. That's very inconvenient for people and quite problematic for automated processes.
- That's why you need to start an SSH agent and pass it the decrypted private key, which it keeps in the main memory and does not give out to everyone. Then, everything will be fine.

However, anyone with the appropriate rights on a machine can prompt any running agent to release the key—or theoretically (as `root`), they can even read the main memory and collect keys there. So, is the agent method now better than leaving the private key unencrypted on the disk?

In operation with a running agent, it's probably not much better because in both cases, the attacker wins as soon as they can assume your identity.

To conclude, in IT practice, the question "Is a solution 100% secure?" is not a serious one. The answer would always be no (except when the machine is turned off). Instead, the crucial questions are these:

- Does the solution offer an acceptable mix of security and convenience?
- Does the solution meet the security obligations expected of employees today?
- Can the solution help prevent any negative findings during a security audit?

We now want to find such a solution with the Ansible Vault.

10.1.2 Getting Started

Let's assume one of our target hosts (e.g., `debian`) requires sensitive information. We'll demonstrate this by using the `host_vars` mechanism (see Chapter 6, Section 6.1.6), ideally with a folder structure for better scalability:

```
$ mkdir -p host_vars/debian && cd $_
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

As an example, you can put the following *vars.yml* file into this folder (the name is arbitrary):

```
---
spoiler: Darth Vader is Luke's father.
```

Listing 10.1 "vars.yml"

Also, you should test right away whether the host "sees" the variable:

```
$ ansible debian -m debug -a var=spoiler
debian | SUCCESS => {
    "spoiler": "Darth Vader is Luke's father."
}
```

Since this information must not fall into the wrong hands, we'll now encrypt it with the `ansible-vault` command. We'll also assign a so-called *Vault-ID* to keep the option to encrypt other files with different passwords later if necessary:

```
$ ansible-vault encrypt --vault-id movie@prompt vars.yml
New vault password (movie):
Confirm new vault password (movie):
Encryption successful
```

Note that `@prompt` means the password should be entered interactively.

Ansible Vault relies on proven technology. The file content is now AES256 encrypted, as you can quickly verify with a glance. All Ansible commands that need access to this file would now fail:

```
$ ansible debian -m debug -a var=spoiler
ERROR! Attempting to decrypt but no vault secrets found
$ ansible all -m ping
ERROR! Attempting to decrypt but no vault secrets found
```

In the simplest case, you'll be prompted for the password each time. You can simply use the same `--vault-id` option that you used for encryption:

```
$ ansible debian -m debug -a var=spoiler --vault-id movie@prompt
```

Of course, you can also use the same mechanism with other commands like `ansible-playbook`, `ansible-console`, etc..

One More Note About Encryption

With `ansible-vault encrypt`, you can encrypt not only variable files but also any other file in an Ansible project: playbooks, role files, Jinja templates, inventories, etc.

The most typical and practical object, however, is variable files.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

10.1.3 Meaning of the Vault ID

The vault ID is a kind of category for the encrypted secret. Technically speaking, the identifier simply ends up in the header area of the encrypted data:

```
$ cat vars.yml
$ANSIBLE_VAULT;1.2;AES256;movie
623566633834376662323737356638643238356666306230 62[…]
626261366161333230376235663436336130653138363338 0a[…]
[…]
```

Adventurous personalities may have already discovered that the vault ID is actually irrelevant when decrypting. The following commands would have also been successful:

```
$ ansible debian -m debug -a var=spoiler --vault-id foo@prompt
$ ansible debian -m debug -a var=spoiler --vault-id @prompt
This one (from the pre-Ansible-2.4 era) is quite classic:
$ ansible debian -m debug -a var=spoiler --ask-vault-pass
```

This wouldn't change even if you work with multiple vault IDs or passwords. Ansible tries each available password in sequence for every encrypted object until one matches. (Well, the third call obviously can't work anymore.)

However, since this default behavior can quickly lead to unclear situations in setups with different vault IDs, I strongly recommend that you always use the following configuration setting:

```
# [defaults]
vault_id_match = true
```

Listing 10.2 "ansible.cfg": Excerpt

As a result, vault IDs must now necessarily match the IDs of the encrypted objects; otherwise, no decryption will be performed.

When there is a technical reason for it, you can easily work with multiple vault passwords, as we've already mentioned. When encrypting another file, you may choose an additional vault ID (e.g., `movie2`). If you need to decrypt with different passwords during a playbook call, you can specify all required vault IDs on the command line:

```
$ ansible-playbook […] --vault-id movie@prompt --vault-id movie2@prompt
Vault password (movie):
Vault password (movie2):
[…]
```

Use cases for different vault IDs include the following:

- Separation of different environments (with the `dev` and `prod` vault IDs)

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

- Separation of teams or departments (with the `developers` and `admins` vault IDs)
- Separation by projects or clients (with the `client_a` and `client_b` vault IDs)

10.1.4 Further Vault Commands

The `ansible-vault` command has, in addition to the `encrypt` subcommand, several other commands that are all quite plausible:

```
Decrypt:
$ ansible-vault decrypt <FILE>

Just view:
$ ansible-vault view <FILE>

Edit (using the EDITOR environment variable, if set):
$ ansible-vault edit <FILE>

Change password:
$ ansible-vault rekey <FILE>
```

10.1.5 A Trick for Finding Variables Again

There is now (at least) one small problem with the encrypted files: the Linux admin's favorite tool—`grep`—no longer works anymore. Until now, we've always answered the question "Where did I define this variable again?" with a quick `grep -r`, but now, it will remain unanswered because there are, of course, no hits in the AES256 data jumble.

I recommend that you use the following workaround (though I don't want to go so far as to call it a best practice): you can leave the variable file unencrypted, and for all secret variables, you can refer to a corresponding variable with the `vault_` prefix using Jinja:

```
---
spoiler: '{{ vault_spoiler }}'
```

Listing 10.3 "vars.yml"

And then, you can introduce a second *vault.yml* file, in which you can actually define all `vault_*` variables:

```
---
vault_spoiler: Darth Vader is Luke's father.
```

Listing 10.4 "vault.yml"

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Now, of course, this file is encrypted. The advantages of this method are as follows:

- You can `grep` for variable names again and at least find the place of definition.
- You can only encrypt truly sensitive data, and you can directly define all noncritical variables in the *vars.yml*.

10.1.6 Encrypting Individual Variables

An alternative to the method we've just shown, which would also leave a file fundamentally readable, is encrypting individual variables. Suppose we overwrite the *vars.yml* file with the following new content:

```
---
title: Star Wars
year: 1977
spoiler: # This is where the secret should go
```

Listing 10.5 "vars.yml"

Only the spoiler should remain a secret. You create the encrypted section interactively:

```
$ ansible-vault encrypt_string --vault-id movie@prompt --stdin-name spoiler
New vault password (movie):
Confirm new vault password (movie):
Reading plaintext input from stdin. (ctrl-d to end input, twice if
  your content does not already have a newline)
Darth Vader is Luke's father.Ctrl+D Ctrl+D
Encryption successful
spoiler: !vault |
          $ANSIBLE_VAULT;1.2;AES256;movie
          32393136663565333962663261366236616362353466326564[…]
          36343866306462393535333664653437616439333036393330a[…]
          33326432623965306166363163373732613537393964353632[…]
          37656264323837363506a393936313330356635663434383462[…]
          31326637623838346663316362636131323936356634383463[…]
```

You can now copy and paste the output (of course, without the `Encryption successful` line) into the variable file.

You could also pass the text to be encrypted via STDIN:

```
$ echo -n "Darth Vader is Luke's father." |
    ansible-vault encrypt_string --vault-id movie@prompt --stdin-name spoiler
```

You could also pass it via parameter, but then, you'd do it with `--name` instead of `--stdin-name`. Please keep in mind that such command lines would also end up in the

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

history of your bash. If that has already happened, there is a remedy—you can use the history command to determine the command number and then remove the command from the history as follows:

```
$ history -d <OFFSET>
$ history -w
```

10.1.7 More Convenience or Automation

Constantly entering decryption passwords is not the ultimate solution from a practical perspective, but there are other options. In addition to interactive input, you can obtain a password from a file or even use an executable program that outputs your password. This also explains the strange syntax with --vault-id; Table 10.1 shows all options.

Parameters	Meanings
<ID>@prompt	Read password interactively.
<ID>@<FILE>	Read password from <FILE>.
<ID>@<PROGRAM>	Read password from the output of <PROGRAM>.

Table 10.1 Options with "--vault-id"

Additionally, you can use the ANSIBLE_VAULT_IDENTITY_LIST environment variable to avoid having to specify all the password sources (and there could be several) on the command line each time. The following simple file-based example can illustrate this.

Warning

We can only halfheartedly recommend the following setup because a sensitive piece of information (the vault password) is stored in a plaintext file. However, the advantage of this setup is that it occurs outside the Ansible project folder, which prevents it from being included in version control!

1. Store your vault password in any file outside the Ansible project, for example, you could use this one:

   ```
   PasswOrd2024
   ```

   Listing 10.6: "~/.vault-pass.movie"

2. Test your "file provider":

   ```
   $ ansible debian -m debug -a var=spoiler \
       --vault-id movie@~/.vault-pass.movie
   ```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

3. Set the environment variable:

   ```
   $ export ANSIBLE_VAULT_IDENTITY_LIST=movie@~/.vault-pass.movie
   ```

   (If you actually have a list of multiple passwords, you should provide them separated by commas.)

4. Now, test without `--vault-id`:

   ```
   $ ansible debian -m debug -a var=spoiler
   ```

Note

Instead of using the `ANSIBLE_VAULT_IDENTITY_LIST` environment variable, you can work with the *vault_identity_list* directive in the `ansible.cfg` (in the `[default]` section). However, the environment variable has higher priority.

10.1.8 Convenience and Relative Security with a Password Client Script

In Table 10.1, you saw that the password provider can also be a program. We refer to such a program as a *password client script*, and you can create one in any programming language. The program file just needs to meet the following requirements:

- The file must be executable.
- The name of the file should end with `-client` or `-client.<SUFFIX>`.
- The program must accept the parameterized `--vault-id` option (automatically passed by Ansible).
- The program must write the corresponding vault password to the standard output.

Now, of course, the question is, "Where does the password come from?" If it were in plaintext in the program code, then the method would obviously be no more secure than the variant with the password file from the previous section. As a first improvement in security, however, the program could, for example, obtain the password from an environment variable—like this Python script from Listing 10.6 does:

```
#!/usr/bin/python3

import os
from argparse import ArgumentParser

parser = ArgumentParser()

parser.add_argument("--vault-id", dest="vault_id",
                    default="none", help="Vault-ID to use")
args = parser.parse_args()
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
vault_id = vars(args)['vault_id']

envar = 'VAULT_PASSWORD_' + vault_id.upper()

if envar in os.environ:
    print(os.environ[envar])
```

Listing 10.6 "~/bin/vault-env-client.py": Simple Password Client Script in Python

It is then used as follows:

```
In this shell, put the vault password into an environment variable:
$ export VAULT_PASSWORD_MOVIE=PasswOrd2024

Test:
$ ansible debian -m debug -a var=spoiler \
    --vault-id movie@~/bin/vault-env-client.py
debian | SUCCESS => {
    "spoiler": "Darth Vader is Luke's father."
}
```

Of course, you can also apply the `ANSIBLE_VAULT_IDENTITY_LIST` environment variable.

10.1.9 Convenient and (Preferably) Secure with GNU Privacy Guard plus Pass

If you want convenience and a bit more security, you need an "agent solution." Ansible doesn't have anything built in here, and it refers to proven additional tools from the Linux toolkit. I would like to introduce you to a solid solution based on GNU Privacy Guard (GPG) and the Linux `pass` password manager.

Please take a short trip to Appendix B. There, you'll learn how to get the password manager up and running. I'll wait here for you!

```
. o 0 o . o 0 o . o 0 o .
```

Now, if you want to store vault passwords in `pass`, you should include the vault ID:

```
pass insert ansible/vault/movie
```

With that, we are almost done. You now have a fairly secure program that can provide vault passwords upon request, and the only small problem is that you can't specify a program with parameters at `--vault-id`. Of course, a small password client script in any programming language can help here again.

The Python solution from Listing 10.7 would be completely sufficient:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
#!/usr/bin/python3

import os
from argparse import ArgumentParser
import subprocess

parser = ArgumentParser()

parser.add_argument("--vault-id", dest="vault_id",
                    default="none", help="Vault-ID to use")
args = parser.parse_args()


vault_id = vars(args)['vault_id']

p = subprocess.run(['pass', f'ansible/vault/{vault_id}'])
```

Listing 10.7 "~/bin/vault-pass-client.py": Wrapper Script for "pass" Password Manager in Python

The final test is as follows:

```
$ ansible debian -m debug -a var=spoiler \
    --vault-id movie@~/bin/vault-pass-client.py
debian | SUCCESS => {
    "spoiler": "Darth Vader is Luke's father."
}
```

10.2 Debugging and Troubleshooting

In this section, you'll find some techniques and strategies that you can apply when issues arise with your playbooks and roles.

10.2.1 Debug Mode and Verbosity Level

In most cases, like a broken connection to a target host or a changed host key, Ansible quickly provides a clear error message that an experienced Linux user can usually understand. However, it becomes more frustrating when Ansible simply hangs without explanation while executing a playbook or ad hoc command. We've already discussed issues with persistent connections in Chapter 3, Section 3.4.2, so let's assume that you've waited sixty seconds and that even after retrying, there's no change.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

In such a situation, it's always a good idea to first enable debugging output by using the `ANSIBLE_DEBUG` environment variable:

```
$ ANSIBLE_DEBUG=1  ansible-playbook […]
```

In addition, or alternatively, you can increase the *verbosity*, which brings even more connection information about the connections to the screen. You can go up to level 4 here, either with the `ANSIBLE_VERBOSITY` environment variable or by repeatedly using the `-v` option:

```
$ ANSIBLE_DEBUG=1 ANSIBLE_VERBOSITY=4  ansible-playbook […]
Or:
$ ANSIBLE_DEBUG=1  ansible-playbook -vvvv […]
```

The challenge now is to extract useful information from this sea of data. Imagine you find something like this in the log before everything comes to a stop:

```
SSH: EXEC ssh [...] /usr/bin/python3 /home/ansible/.ansible/tmp/
     ansible-tmp-1588862715.5502043-251744962800523/AnsiballZ_setup.py
```

There is a hang in the gathering facts phase (in the `setup` module), but you probably already knew that. If you're an absolute expert, you could now log in to the target system, manually execute the corresponding command, and follow it with `strace`. If the temporary files are already gone, you can start the whole thing again with `DEFAULT_KEEP_REMOTE_FILES=1` to prevent any cleanup operations. In the `strace` output, you might find something like this:

```
stat("/etc/ansible/facts.d/test.fact", {st_mode=S_IFREG|0755, [...]
[...]
wait4
wait4
wait4
[...]
```

So, there seems to be a problem with this self-created dynamic fact program. All of this was, of course, staged, but the diagnostic possibilities are often helpful in situations like this.

10.2.2 Improving the Readability of Outputs

When a task fails, Ansible by default provides an error output formatted in JSON, which some find difficult to read. For example, the following task would result in an error (because you should use the `shell` module instead of the `command` module, due to the pipe):

```
    - name: Count processes
      command: ps -ef | wc -l
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

The corresponding error message when executing this task looks something like this:

```
[…]
TASK [Count processes] *********************************************************
fatal: [localhost]: FAILED! => {"changed": true, "cmd": ["ps", "-ef", "|",
"wc", "-l"], "delta": "0:00:00.003348", "end": "2024-04-16 08:54:26.059880"
, "msg": "non-zero return code", "rc": 1, "start": "2024-04-16 08:54:26.056
532", "stderr": "error: garbage option\n\nUsage:\n ps [options]\n\n Try 'ps
 --help <simple|list|output|threads|misc|all>'\n  or 'ps --help <s|l|o|t|m|
a>'\n for additional help text.\n\nFor more details see ps(1).", "stderr_li
nes": ["error: garbage option", "", "Usage:", " ps [options]", "", " Try 'p
s --help <simple|list|output|threads|misc|all>'", "  or 'ps --help <s|l|o|t
|m|a>'", " for additional help text.", "", "For more details see ps(1)."],
"stdout": "", "stdout_lines": []}
[…]
```

Thanks to so-called *callback plug-ins* (which you will learn more about in Chapter 16), you can customize this output. Specifically, you'll have various output plug-ins available, and you can select them via the `ANSIBLE_STDOUT_CALLBACK` environment variable or by Ansible configuration. (The former option is stronger.) We highly recommend that you use the YAML output plug-in in this case, as it provides much more readable output. You can set it up in the configuration file as follows:

```
# [defaults]
stdout_callback = yaml
```

Listing 10.8 "ansible.cfg": Excerpt for Activating YAML Output Plug-in

Or you can start a playbook with it *on the fly*:

```
$ ANSIBLE_STDOUT_CALLBACK=yaml ansible-playbook […]
```

If the above task fails, the output will now look much better:

```
[…]
TASK [Count processes] *********************************************************
fatal: [localhost]: FAILED! => changed=true
  cmd:
  - ps
  - -ef
  - '|'
  - wc
  - -l
  delta: '0:00:00.003377'
  end: '2024-04-16 09:08:59.039604'
  msg: non-zero return code
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
  rc: 1
  start: '2024-04-16 09:08:59.036227'
  stderr: |-
    error: garbage option

    Usage:
     ps [options]

     Try 'ps --help <simple|list|output|threads|misc|all>'
      or 'ps --help <s|l|o|t|m|a>'
     for additional help text.

    For more details see ps(1).
  stderr_lines:
  - 'error: garbage option'
  - ''
  - 'Usage:'
  - ' ps [options]'
  - ''
  - ' Try ''ps --help <simple|list|output|threads|misc|all>'''
  - '  or ''ps --help <s|l|o|t|m|a>'''
  - ' for additional help text.'
  - ''
  - For more details see ps(1).
  stdout: ''
  stdout_lines: <omitted>
[…]
```

In Chapter 16, Section 16.1, you'll find many more ways to change the output. In my opinion, the YAML plug-in offers the best compromise between readability and structured information.

Important

The configured output plug-in affects not only error messages but also the normal output. However, at least with the YAML format, the normal output also gains in readability, so that should not be a problem.

Just keep an eye out in case the output of your playbooks suddenly looks different.

10.2.3 Gathering Facts Takes Too Long

In most projects, you'll need host facts, which are collected at the beginning of a play by default. We've already discussed that you should cache these facts for performance reasons (see Chapter 6, Section 6.3.2), but every cache expires eventually, and then, fact

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

gathering starts again. If you find at that moment that there are intolerable delays, you could limit the amount of gathered facts. You'll rarely need all of them—so why collect information about disks or network interfaces if they're not relevant in any of your playbooks?

You can control the gathering on the play level with the `gather_subset` keyword, which expects a list of subsets to include or exclude as a value. The default is `all`, and `min` is somewhat the opposite. You gather nothing with `!all`, `!min`. (The exclamation mark is used for negation.) The following example shows how to collect only distribution information for all your systems:

```
- hosts: all

  gather_facts: yes
  gather_subset:
    - '!all'
    - '!min'
    - distribution
```

Listing 10.9 Example Header Section of Playbook

The available set names are listed on the documentation page for the setup module (online or offline with `ansible-doc setup`). Names like `interfaces` or `processor` should also suffice as explanations.

To summarize, first, you need to try to find out which facts are causing the delays in your case, and then, you need to exclude them by using `gather_subset` *if* you don't need them.

10.2.4 The Playbook Debugger

Ansible comes with a built-in debugger for intervening in running playbooks. For example, you can inspect tasks, check or change variables or module parameters, and restart failed tasks.

Let's take this buggy playbook and see what the debugger can (and cannot) do with it:

```
---
- hosts: debian

  tasks:
    - name: Startmessage
      debug: msg="Here we go!"

    - name: Copy file to /tmp
      copy:
        dest: /tmp
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
        content: |
          Content of the file

    - name: Count running processes
      command: ps -ef | wc -l
      register: ps_count_cmd

    - debug: var=ps_count_cmd.stdout
```

Listing 10.10 "debugtest.yml": Playbook Full of Problems

A normal run of the playbook would fail at the `copy` task. Can you see why? In this case, a *filename* (and not just a directory name) must be specified at the destination.

By setting the `ANSIBLE_ENABLE_TASK_DEBUGGER` environment variable, you can enable the debugger as soon as a task returns `failed` or `unreachable`:

```
$ ANSIBLE_ENABLE_TASK_DEBUGGER=1  ansible-playbook debugtest.yml
PLAY [debian] ******************************************************************


TASK [Startmessage] ************************************************************
ok: [debian] => {
    "msg": "Here we go!"
}

TASK [Copy file to /tmp] *******************************************************
fatal: [debian]: FAILED! => {"changed": false,
  "msg": "cannot use content with a dir as dest"}

[debian] TASK: Copy file to /tmp (debug)> help

Documented commands (type help <topic>):
========================================
EOF  c  continue  h  help  p  pprint  q  quit  r  redo  u  update_task
```

Those are all the available options, and what can be printed is listed in the online documentation and Table 10.2.

Arguments	Meanings
`task`	The name of the current task
`host`	The name of the current target hosts
`result._result`	The current result as a data structure

Table 10.2 Print Options in Debugger

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Arguments	Meanings
`task.args`	The module parameters
`task_vars`	All variables that are visible to the task

Table 10.2 Print Options in Debugger (Cont.)

You can also omit the `p` as a command; for example, `host` works just as well as `p host`.

The key takeaway here is that `task.args` and `task_vars` are writeable. Now, let's fix the error with the copy destination and restart the task:

```
[debian] TASK: Copy file to /tmp (debug)> task.args['dest'] = '/tmp/test1'
[debian] TASK: Copy file to /tmp (debug)> r
changed: [debian]

TASK [Count running processes] ********************************************
fatal: [debian]: FAILED! => {"changed": true, "cmd": ["ps", "-ef", "|", […]
```

And now . . . we immediately encounter the next faulty task. (Do you see the problem with this one too? We should have used the `shell` module.) Unfortunately, the debugger doesn't help here anymore, as you can change the module parameterization but not the module itself. So, at this point, we give up and exit the environment with `q`.

Further Options to Start the Debugger

If you want to involve the debugger in things beyond `failed` and `unreachable` tasks, you can use the `debugger` keyword at any sensible level: play, role, block, or task. (This means you need to change your playbook or role!) If it occurs on multiple levels simultaneously, the more specific definition wins. The following values are possible:

- **`Always`**
  Start the debugger for every task.
- **`Never`**
  Never start the debugger.
- **`on_failed`**
  Start only on task failure.
- **`on_unreachable`**
  Start only when a target host is unreachable.
- **`on_skipped`**
  Start only when the task is skipped.

Author's Conclusion and Opinion

If you think you need the debugger, then go ahead and use it. Personally, in twenty-five years of programming, I've never needed a debugger, and it's no different with

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Ansible—especially since, as you've just seen, you won't get far with certain errors anyway.

Tip

Follow Linus Torvalds example: don't use a debugger—get it right from the start!

10.2.5 Static Code Analysis with "ansible-lint"

Linting in software development means statically analyzing code to find issues that a compiler (in our case, Ansible) wouldn't flag. The term *lint* is fitting because just like you would remove lint from your clothing once you notice it, linting tools help identify small, often overlooked issues in code.

The Ansible team provides an additional tool for this called `ansible-lint`. You can install it quickly (either with root privileges or in a virtualenv environment):

```
$ sudo pip3 install ansible-lint
```

Let's check our buggy playbook from the last section with it:

```
$ ansible-lint debugtest.yml
WARNING  Listing 10 violation(s) that are fatal
name[play]: All plays should be named.
debugtest.yml:2

fqcn[action-core]: Use FQCN for builtin module actions (debug).
debugtest.yml:5 Use `ansible.builtin.debug` or `ansible.legacy.debug`
                instead.

no-free-form: Avoid using free-form when calling module actions. (debug)
debugtest.yml:5 Task/Handler: Startmessage

fqcn[action-core]: Use FQCN for builtin module actions (copy).
debugtest.yml:8 Use `ansible.builtin.copy` or `ansible.legacy.copy`
                instead.

risky-file-permissions: File permissions unset or incorrect.
debugtest.yml:8 Task/Handler: Copy file to /tmp

fqcn[action-core]: Use FQCN for built-in module actions (command).
debugtest.yml:14 Use `ansible.builtin.command` or `ansible.legacy.command`
                 instead.

no-changed-when: Commands should not change things if nothing needs doing.
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
debugtest.yml:14 Task/Handler: Count running processes

fqcn[action-core]: Use FQCN for built-in module actions (debug).
debugtest.yml:18 Use `ansible.builtin.debug` or `ansible.legacy.debug`
                 instead.

name[missing]: All tasks should be named.
debugtest.yml:18 Task/Handler: debug var=ps_count_cmd.stdout

no-free-form: Avoid using free-form when calling module actions. (debug)
debugtest.yml:18 Task/Handler: debug var=ps_count_cmd.stdout

Read documentation for instructions on how to ignore
specific rule violations.

                      Rule Violation Summary
 count tag                    profile    rule associated tags
     2 no-free-form           basic      syntax, risk
     1 name[missing]          basic      idiom
     1 name[play]             basic      idiom
     1 risky-file-permissions safety     unpredictability
     1 no-changed-when        shared     command-shell, idempotency
     4 fqcn[action-core]      production formatting

Failed: 10 failure(s), 0 warning(s) on 1 files.
Last profile that met the validation criteria was 'min'.
```

Here, the tool has found an impressive ten "fatal violations"—but none of them are actual problems with the playbook. Well, you probably shouldn't expect miracles here, so feel free to try our Apache playbook from Chapter 6, Section 6.5.5, as a more challenging input:

```
$ ansible-lint apache3.yml
WARNING  Listing 18 violation(s) that are fatal
[…]
```

The "serious" problems here are essentially that no FQCNs were used at all, there are no spaces at the beginning and end of the `{{ ... }}` environment when accessing variables with Jinja, and instead of `true`/`false`, the `yes`/`no` identifiers were used. This is more a matter of taste and aesthetics, but it doesn't affect the functionality of the playbook in the slightest.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Preventing Certain Rules from Being Applied

If you don't like certain linting rules, you can ignore them with the `-x` option. You can find the corresponding identifiers in the `Rule Violation Summary` at the end of the linting result. Here's an example:

```
$ ansible-lint -x "fqcn,name[play]" debugtest.yml
```

If you'd like to always exclude certain rules, you can also specify this in a configuration file. By default, `ansible-lint` searches for the *.ansible-lint* file in the current working directory, but with the `-c` option, you can also provide any path name:

```
skip_list:
  - fqcn
  - name[play]
```

Listing 10.11 ".ansible-lint": Example of "ansible-lint" Configuration File

As a last resort, you could include special comments in the form `# noqa [...]` in a specific task if you want to selectively skip the application of certain rules, as in the following example:

```
    - name: Count running processes
      shell: ps -ef | wc -l
      register: ps_count_cmd
      # noqa: no-changed-when
```

Listing 10.12 Example of Task with Reduced Linting Rules

This would mean that the `no-changed-when` rule would not apply to this task.

You can find more information at *https://ansible.readthedocs.io/projects/lint*. In general, it's worth diving deeper into the documentation if you plan to use the tool seriously. The offline documentation, which is accessible via `ansible-lint -h`, is also worth exploring!

Conclusion

It certainly doesn't hurt to check your playbooks or roles with `ansible-lint`. If you occasionally disagree with the tool, you can use exclusion lists or hint comments (as shown).

However, don't expect miracles—most serious issues arise at runtime, and they're difficult to predict without a crystal ball!

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

10.2.6 Check Mode and Diff Mode

In Ansible, there are two modes of operation that are particularly helpful during development: check mode and diff mode. You can use these independently or together.

Check Mode

In Ansible, *check mode* is a kind of test run that only simulates the execution of a playbook, without actually making changes to target systems. You activate such a dry run with the `--check` option.

It's inevitable that not all modules will be able to predict with 100% accuracy what will happen in the event of a failure. This works reliably with modules like `file`, `copy`, and `lineinfile`, but `command` or `shell` calls cannot be simulated at all (in check mode, such tasks are simply skipped). In the relevant module documentation, you will usually find a section indicating whether the module can fully, partially, or simply not support check mode.

You can also control check mode with the `check_mode` directive and then query the current state with the implicit `ansible_check_mode` variable. You can find examples of this at *https://docs.ansible.com/ansible/latest/user_guide/playbooks_checkmode.html*.

Diff Mode

Manipulating file contents is a common source of errors, especially when done with modules like `lineinfile` and `replace`. These tasks often involve tricky regular expressions, which, as we know, can be difficult to manage. Consider this playbook, which is designed to enable SSH access for the root user using a password on the Debian host of our lab environment. (This is just an example!)

The current state in the */etc/ssh/sshd_config* is this:

```
#PermitRootLogin prohibit-password
```

Listing 10.13 "/etc/ssh/sshd_config"

The following playbook is supposed to make the adjustment. (I am omitting the full version with a handler to reload the service here.):

```
---
- hosts: debian

  tasks:
    - name: Modify sshd_config
      lineinfile:
        path: /etc/ssh/sshd_config
        line:   'PermitRootLogin yes'
        regexp: '^#PermitRootLogin'
```

Listing 10.14 "sshd1.yml": Changing SSH Configuration, Version 1

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

If you're not completely sure, you can run the playbook in diff mode to make the changes visible onscreen. Additionally, enabling check mode is, of course, extremely useful so that the changes are only simulated at first:

```
$ ansible-playbook sshd1.yml --check --diff
TASK [Modify sshd_config] ***********************************************
--- before: /etc/ssh/sshd_config (content)
+++ after: /etc/ssh/sshd_config (content)
@@ -30,7 +30,7 @@
 # Authentication:

 #LoginGraceTime 2m
-#PermitRootLogin prohibit-password
+PermitRootLogin yes
 #StrictModes yes
 #MaxAuthTries 6
 #MaxSessions 10

changed: [debian]
```

That looks very good, and on your screen, it will even be shown in very readable color. But what if the original line had not been commented out, like this?:

```
PermitRootLogin prohibit-password
```

Listing 10.15 "/etc/ssh/sshd_config"

In this case, I suggest you modify the target system to simulate this and then do another test run:

```
$ ansible-playbook sshd1.yml --check --diff
TASK [Modify sshd_config] ***********************************************
--- before: /etc/ssh/sshd_config (content)
+++ after: /etc/ssh/sshd_config (content)
@@ -122,3 +122,4 @@
 #      ForceCommand cvs server
 UseDNS no
 GSSAPIAuthentication no
+PermitRootLogin yes

changed: [debian]
```

This time, the `regexp` expression did not match, so `lineinfile` simply appends the desired line to the end of the file—which is certainly not ideal. A better version would be this:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts: debian

  tasks:
    - name: Modify sshd_config
      lineinfile:
        path: /etc/ssh/sshd_config
        line:   'PermitRootLogin yes'
        regexp: '^#?\s*PermitRootLogin'
```

Listing 10.16 "sshd2.yml": Changing SSH Configuration, Version 2

This means the number sign (#) at the beginning of the line is optional, and even zero or more spaces are allowed before `PermitRootLogin`.

So, if possible, you should first use `--check` and `--diff` if you aren't completely sure about content file manipulations—and maybe even if you are. If you still need to catch up on regular expressions, you can take a look at Appendix D.

10.2.7 Last but Not Least: The "debug" Module

You can use the `debug` module to check the contents of variables during the execution of a playbook. It doesn't always have to be as elaborate as shown here:

```
- debug: msg="The value of number is {{number}}"
```

For quick debugging in between, this is enough:

```
- debug: var=number
```

The fact that the `msg` parameter also accepts a list is quite useful, if you want multiline output:

```
- debug:
    msg:
      - "First line"
      - "Second line"
```

The useful `verbosity` parameter takes a value from `0` to `4`, which activates the output only if at least the corresponding number of `-v` switches are passed on the command line. However, only values up to `2` are of practical use because `ansible-playbook` generates too much additional output at higher debug levels, which causes your own debug messages to get lost.

Alternatively, you can always link a `debug` task with a `when` condition to see the message only under certain circumstances. That way, you can easily build your own debug on-off switch:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- hosts: localhost

  tasks:
    - debug: msg="This is always displayed"

    - debug:
        msg: "This is only displayed if mydebug is set"
      when: mydebug is defined
```

Listing 10.17 "mydebug.yml": Debug Messages on Demand

If you want to see such messages, you should make sure to assign a value to the `mydebug` variable, like this:

```
$ ansible-playbook mydebug.yml -e mydebug=1
[…]
ok: [localhost] => {
    "msg": "This is only displayed if mydebug is set"
}
[…]
```

The `debug` name was not chosen without reason for this module. (Otherwise, the Ansible creator could have called it "print."). All in all, it's probably the most frequently used debugging option among Ansible users.

10.3 Accelerating Playbooks with Pipelining

There's a way to speed up the workflow of Ansible a bit. You might remember what Ansible usually does when calling a module. (I outlined this in Chapter 3, Section 3.4.3.) It generates an executable script, copies it to the target host, and then executes it there.

The clever idea now is to skip the copying step and execute the program in one step through a pipe. This technique is called *pipelining*, and you can activate it in the Ansible configuration or via an environment variable:

- Here's how you activate it in the Ansible configuration (note the INI section `ssh_connection`):

  ```
  [ssh_connection]
  pipelining = true
  ```

  Listing 10.18 "ansible.cfg": Excerpt for Enabling Pipelining

- And here's how you activate it via an environment variable:

  ```
  ANSIBLE_PIPELINING=true
  ```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

But of course, there's a catch: it doesn't work in conjunction with `sudo` if the *requiretty* directive is active in the `/etc/sudoers` file on a target system. For this reason, pipelining is disabled by default. However, you can easily test it, because if there is an issue, a related error message will simply appear. (By the way, in our lab environment, there will not be a problem.)

What your actual performance gain will be very much depends on your playbook. I can't give you a general estimate—at the very least, your performance environment shouldn't become slower. My conclusion is that you should try it and be happy if you get an improvement!

10.4 The Talking Cow

When you install the *cowsay* distribution package, you automatically enable one of Ansible's most important features—the talking cow:

```
$ ansible-playbook hello-ansible.yml
 ------------------
< PLAY [localhost] >
 ------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

 --------------
< TASK [debug] >
 --------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

ok: [localhost] => {
    "msg": "Hello Ansible!"
}


[…]
```

If you (incomprehensibly) do not want to see cows despite having `cowsay` installed, you can disable it via Ansible config with `nocows=true`.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

There's also the `cow_selection` directive, which allows you to activate other cow colleagues (see `cowsay -l`). The `random` value is also available. To prevent things from getting too wild, you can define a comma-separated list of cow colleagues you would like to see by using `cowsay_enabled_stencils`, like this:

```
# [defaults]
cow_selection = random
cowsay_enabled_stencils = default, koala, tux
```

Listing 10.19 "ansible.cfg": Excerpt for Random Selection of Cow Colleagues

10.5 Ansible in Pull Mode

As you know, Ansible works in push mode by default, meaning the control host initiates the connections to the targets and solely determines when actions are performed there.

However, this method is not ideal in all cases. For instance, the following situations present problems:

- Certain target hosts are only intermittently online, making them difficult to "catch."
- Certain target hosts are unreachable via the standard network path.
- Certain target hosts need to be set up or provisioned for the first time and are not yet listed in any inventory.

In these cases, a *pull method* is more desirable. Ansible fills this gap with the `ansible-pull` tool, which we'll take a closer look at now.

10.5.1 "ansible-pull": Technique and Requirements

Each time `ansible-pull` is called, a repository from a version control system (usually Git) is checked out, and then a playbook contained within it is started. That's basically all there is to it. In particular, you should not expect an integrated service (i.e., "pull agent"). All necessary automation is usually implemented via Cron.

An example should answer any of your questions. Suppose we want to regularly equip our Ubuntu target host with a new message-of-the-day notification, so that when you log in, you get something like this:

```
 ______________________________________
/ In matters of taste, there can be no \
| disputes.                            |
|                                      |
\ -- Immanuel Kant                     /
 --------------------------------------
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
\   ^__^
 \  (oo)\_______
    (__)\       )\/\
        ||----w |
        ||     ||
```

First, you need to ensure the necessary prerequisites on the target host as follows:

1. Log in there:

   ```
   $ ssh ubuntu
   ```

2. Install the *ansible* and *git* packages:

   ```
   $ sudo apt install ansible git
   ```

3. Ensure that the name of our Git server can be resolved (which is also only necessary in our lab). An entry in */etc/hosts* like this will suffice for the moment:

   ```
   192.168.150.100  gitea.example.org
   ```

   Listing 10.20 "/etc/hosts"

10.5.2 Getting Started

For this example, you'll complete the following steps to set up a very simple Git repository on our Git(ea) server, which is initially empty. (Is the server still running? If not, check Chapter 9, Section 9.4 or Section 9.8 and restart it.)

1. Create a repository named `pull-motd` in the Gitea web application. As the owner, you choose the `ansible` organization as usual. Otherwise, you don't need any special features.
2. Switch to the command line of your control host and create a corresponding project directory locally, like this one:

   ```
   $ mkdir -p ~/ansible/pull/motd && cd $_
   ```

3. Place the following file there:

   ```
   ---
   - hosts: localhost

     tasks:
       - debug: msg="Hello ansible-pull!"
   ```

   Listing 10.21 "local.yml"

4. Initialize the repository and push the commit:

   ```
   $ git init
   $ git add .
   ```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
$ git commit -m "Initial commit"
$ git remote add origin http://gitea.example.org/ansible/pull-motd.git
$ git push -u origin main
```

Then, you'll test on the target host. On the Ubuntu target system, you'll test as follows:

```
$ sudo ansible-pull -U http://gitea.example.org/ansible/pull-motd.git
[WARNING]: Could not match supplied host pattern, ignoring: ubuntu
localhost | SUCCESS => {
    "after": "a4dd6557e216cbb1a331bb3e89bcd100586d9a42",
    "before": "a4dd6557e216cbb1a331bb3e89bcd100586d9a42",
    "changed": false,
    "remote_url_changed": false
}
[WARNING]: No inventory was parsed, only implicit localhost is available.
[WARNING]: Provided hosts list is empty, only localhost is available.
Note that the implicit localhost does not match 'all.'
[WARNING]: Could not match supplied host pattern, ignoring: ubuntu.

PLAY [localhost] ***************************************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [debug] *******************************************************************
ok: [localhost] => {
    "msg": "Hello ansible-pull!"
}

PLAY RECAP *********************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0
                       skipped=0    rescued=0    ignored=0
```

This executes the following steps: `ansible-pull` checks out the specified repository, looks for a Playbook named *local.yml*, and starts it. You can safely ignore all warnings; they are related to the fact that we have not specified an inventory; we are only concerned with the always available `localhost` anyway. If you want to see what's happening behind the scenes, you can debug it as follows:

```
$ ANSIBLE_DEBUG=1 sudo -E ansible-pull -U […] | grep EXEC
```

10.5.3 The Full Solution

We want to see talking cows when we log in. For this, we need to spice up the *local.yml* a bit:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts: localhost

  tasks:
    - name: Update package index
      apt:
        update_cache: yes
        cache_valid_time: 3600

    - name: Install required packages
      apt:
        name:
          - cowsay
          - fortunes-de

    - name: Generate motd text
      shell: /usr/games/fortune de | /usr/games/cowsay
      register: fortune_cmd

    - name: Write into /etc/motd
      copy:
        dest: /etc/motd
        content: |
          {{ fortune_cmd.stdout }}
```

Listing 10.22 "local.yml": Example Playbook for "ansible-pull" Operation

Also, you should add a small configuration file so that Ansible or `ansible-pull` doesn't constantly annoy you with cows because of the installed `cowsay`:

```
[defaults]
nocows = true
```

Listing 10.23 "ansible.cfg"

Then, you put this into Git:

```
$ git add .
$ git commit -m "New version" && git push
```

Next, you test again on the Ubuntu system to see if everything works as expected. We've saved ourselves from creating a cron job, but you can do it if you like.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

10.5.4 What Might Still Be Missing?

Here are a few quick tips before we leave this topic:

- If you're bothered by the warnings, you can add the following option when calling `ansible-pull`:

  ```
  $ ansible-pull […] -i $(hostname),
  ```

- If you want to use a specific branch in the repository, then this helps:

  ```
  $ ansible-pull […] -C <BRANCH>
  ```

- If the playbook should only run if it has changed, you should use this:

  ```
  $ ansible-pull […] -o
  ```

- If you want to access a private repository with SSH key authentication, then this helps:

  ```
  $ ansible-pull […] --private-key <KEYFILE>
  ```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 11
Ansible and Docker

Software containers have probably been the most influential IT technology in recent years, and the name *Docker* is still inextricably linked with this technology. Even though there are now alternatives like *Podman* that undoubtedly have their advantages, people still often use the well-established original.

In this chapter, I would like to show you ways to combine the strengths of Docker and Ansible, and I'd also like to explain how you can compensate for some of Docker's weaknesses by using Ansible. I assume that you have some knowledge in the Docker environment; in particular, you should be familiar with the Docker command line and know terms like *image*, *container*, *Docker hub*, *user-defined network*, *volume*, *Dockerfile*, and *Docker Compose*.

If you are interested in this topic but haven't had any experience with Docker yet, you should first read up on the subject and try out various things at your own pace. For an introduction, I can recommend *Docker: Practical Guide for Developers and DevOps Teams* by Bernd Öggl and Michael Kofler (Rheinwerk Computing, 2023, *www.rheinwerk-computing.com/5650*) and online resources such as *https://docker-curriculum.com/*. There are countless good contributions on YouTube as well; just search for "Docker Tutorial."

11.1 Installing Docker

Equipping a Linux system with Docker is not rocket science, but it's also not entirely trivial. Since I don't want to bore you with details here, you should feel free to use my CBI tool, which makes this task much easier. You've encountered this tool before if you've worked through Chapter 9 on Ansible web interfaces.

To not give you the impression that Ansible and Docker must always be on the same machine, we want to make one of the four target hosts the Docker host for all further considerations in our lab. If you already have one or more Docker (test) hosts outside the lab environment, you can skip the following description and proceed directly to Section 11.2.

The Docker setup procedure using CBI works with all four operating systems we are considering; here, I will use the Ubuntu machine as an example. These are the steps you need to follow:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

1. Log in to the target system:

   ```
   $ ssh ubuntu
   ```

2. Install Ansible and Git there:

   ```
   $ sudo apt install ansible git
   ```

3. Clone my GitHub repository and then switch to the cloned project folder:

   ```
   $ git clone https://github.com/ansible-buch/cbi.git
   $ cd cbi
   ```

4. Create a configuration file:

   ```
   $ ./cbi config
   ```

5. Start the installer with admin rights.

   Then, you pass the `user` variable the username of the unprivileged account you are currently working with:

   ```
   $ sudo ./cbi up docker/environment -e user=vagrant
   ```

Now, you could remove the Ansible package again because the Docker host is just a normal target host where Ansible does not need to be present. However, you should leave it installed for later demonstrations because we'll need it when the control host and Docker host are the same machine.

What does the setup procedure do, and what is now available to you?

- First, it performs an operating system–specific installation of the Docker Engine (which was formerly known as Docker Community Edition). It uses the official package sources from `docker.com` (except in the case of openSUSE, as Docker, Inc. does not maintain packages for this distribution). You can find the documentation for the sources at *https://docs.docker.com/engine/install/*.
- It starts the Docker service and integrates it into the autostart. It also uses a minimal configuration file to set up log rotation to prevent the hard drive from filling up due to Docker's internal logging.
- It adds the user set by the Ansible variable the system group Docker so that this account can also execute Docker commands. This isn't important for our Ansible context; it's just convenient if Docker commands are to be executed directly on the machine and without Ansible.
- It installs the `lazydocker` tool directly from its GitHub repository. This is a very useful terminal-based management tool for Docker, and while it's not needed or required for our considerations, you should try it out once you know your way around Docker.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

11.2 Docker Modules

In this section, we'll introduce the concept of Docker modules and then walk through a list of the most useful modules.

11.2.1 Preparations and Preliminary Considerations

All Docker modules are based on the same mechanics:

- A module is to be executed. For this, the Ansible control host addresses a target host (as usual).
- This target host *can* be the Docker host (that would be the default), but it doesn't have to be. The Docker client software, which must be on this target, could now also address another machine—namely, the actual Docker host—via the Docker application programing interface (API).

Personally, I would avoid having the target host and docker host be on different machines. The effort on the Docker side alone to securely provide the API over the transmission control protocol (TCP) or internet protocol (IP) is not insignificant. However, if someone else has already set it up for you and you just have to "simply" use it, then you should know that all Docker modules provide the following parameters:

- **`docker_host`**
  This is a URL or socket path to contact the Docker Host.
  Environment variable: `DOCKER_HOST`
  Default: `unix://var/run/docker.sock`
- **`tls_hostname`**
  This is the *expected* server name for certificate validation.
  Environment variable: `DOCKER_TLS_HOSTNAME`
  Default: `localhost`
- **`api_version`**
  This is the version of the Docker API
  Environment variable: `DOCKER_API_VERSION`
  Default: `auto`
- **`client_cert` or `cert_path`**
  This is the directory path to the client certificate.
  Environment variable: `DOCKER_CERT_PATH`
  Default: `(none)`
- **`ssl_version`**
  This is the SSL version to be used.
  Environment variable: `DOCKER_SSL_VERSION`
  Default: (from the Python module *ssl.py*)

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

- **tls**
  You use this to determine whether to use transport layer security (TLS).
  Environment variable: `DOCKER_TLS`
  Default: `no`
- **validate_certs or tls_verify**
  You use one of these to determine whether to validate certificates for their authenticity?
  Environment variable: `DOCKER_TLS_VERIFY`
  Default: `no`
- **timeout**
  You use this to determine how long (in seconds) the maximum wait for a response from the Docker host should be.
  Environment variable: `DOCKER_TIMEOUT`
  Default: `60`

Important

By using environment variables, you can cleverly avoid having to specify the same four to six parameters for every Docker-related task. However, you have to set these environment variables on the *target host*! If your setup consists of an Ansible control host, Ansible target host, and Docker host, you can probably do the best job of specifying the variables in an `environment` section of the playbook (see Chapter 6, Section 6.14).

An alternative to environment variables would be to use the `module_defaults` module (see Chapter 19, Section 19.4.3).

11.2.2 A First Simple Example

Perhaps the previous section brought you more confusion than clarity—so let's take a look at Listing 11.1, which describes a typical situation in which the target host and Docker host are the same machine:

```
---
- hosts: ubuntu

  tasks:
    - name: Run the hello-world container
      docker_container:
        name: hello-world
        image: library/hello-world
        detach: false
      register: run
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    - debug: msg={{ run.container.Output[:39] + '[...]' }}

    - name: Remove container
      docker_container:
        name: hello-world
        state: absent
```

Listing 11.1 "hello-docker.yml": Starting and Removing Docker Container

This playbook simply starts the usual `hello-world` container, and I've used Jinja to abbreviate the long output. Once the container has served its purpose, it is deleted again. This is how it looks when called:

```
$ ansible-playbook hello-docker.yml
[…]
TASK [debug] *******************************************************************
ok: [ubuntu] => {
    "msg": "\nHello from Docker!\nThis message shows [...]"
}
[…]
```

11.2.3 Overview

You can find a list of Docker modules that are available in Ansible at *https://docs.ansible.com/ansible/latest/collections/community/docker/*. I'll give you a brief overview here, and I want to make it clear that it's not complete—in particular, I omit modules that can only be used with *Docker Swarm*. (They are not worse than or different from the single-node modules in any way, but Docker Swarm is simply not very common anymore. If you use Docker Swarm and want to manage a swarm with Ansible, you'll be able to handle it effortlessly.)

From the documentation entry page, you can already see that all modules are part of the `community.docker` collection. For simplicity's sake, as usual, I will refrain from constantly specifying the FQCN.

"docker_host_info": Querying Host Information

```
- name: Query host information
  docker_host_info:
    containers: no
    disk_usage: no
    images: no
    networks: no
    volumes: no
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
  register: result

- debug: var=result
```

With this module, you can obtain information that a `docker system info` would provide. Additionally, you can get information about containers, disk usage, images, networks, and volumes if you activate the corresponding parameters in the code example above. By default, these are all set to "no," but I have included them in the listing for documentation purposes.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_host_info_module.html*.

"docker_container": Managing Containers

```
- name: Start Apache container
  docker_container:
    name: apache1
    image: httpd
    published_ports:
      - '8001:80'
```

With the `docker_container` module, you can manage the entire lifecycle of a container. This module's capabilities are roughly equivalent to those of the `docker container` command, so the number of available parameters is quite long.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_container_module.html*. Note the really large number of examples there!

"docker_container_info": Querying Container Information

```
- name: Query container information
  docker_container_info:
    name: apache1
  register: result

- debug: var=result
```

This module collects the information that a `docker container inspect` would give you.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_container_info_module.html*.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

"community.docker.docker_container_exec": Executing Commands in Containers

```
- name: Run command
  community.docker.docker_container_exec:
    container: apache1
    command: echo Hello Apache
  register: result

- debug: var=result.stdout
```

This module behaves like a `docker container exec` would.

Important

The `docker_container_exec` module was only added in `community.docker` version 1.5.0. It's not a "legacy" module, and it therefore must be fully qualified when called!

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_container_exec_module.html*.

"docker_network": Managing User-Defined Networks

```
- name: Create user-defined network
  docker_network:
    name: testnet1

- name: Remove network
  docker_network:
    name: testnet1
    state: absent
```

With the `docker_network` module, you can manage Docker networks and connect containers to them. In terms of its capabilities, it roughly corresponds to those of the `docker network` command.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_network_module.html*.

"docker_network_info": Querying Network Information

```
- name: Query network information
  docker_network_info:
    name: bridge
  register: result

- debug: var=result
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

The capabilities of this module reflect those of the `docker network inspect` command.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_network_info_module.html*.

"docker_volume": Managing Volumes

```
- name: Create volume
  docker_volume:
    name: data1

- name: Remove volume
  docker_volume:
    name: data1
    state: absent
```

With this module, you can create and delete Docker volumes. It serves the same function as a `docker volume` command.

The online documentation for it is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_volume_module.html*.

"docker_volume_info": Querying Volume Information

```
- name: Query volume information
  docker_volume_info:
    name: data1
  register: result

- debug: var=result
```

With the `docker_volume_info` module, you can query information about volumes. In terms of its capabilities, it corresponds to the `docker volume inspect` command.

The online documentation for it is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_volume_info_module.html*.

"docker_login": Managing Registry Logins

```
- name: Log in to private registry
  docker_login:
    registry: hub.example.org
    username: john
    password: secret
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
- name: Log out
  docker_login:
    registry: hub.example.org
    state: absent
```

With the `docker_login` module, you can authenticate with private or public registries to pull images from them or push images to them, depending on what you're authorized to do. It corresponds to a `docker login` command.

Note that (also completely analogous to `docker login`) the login credentials are stored on the target host in the *~/.docker/config.json* file.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_login_module.html*.

"docker_prune": Performing Cleanup Operations

```
- name: Throw away everything that is no longer needed
  docker_prune:
    containers: yes
    images: yes
    networks: yes
    volumes: yes
    builder_cache: yes
```

This module provides a unified interface with the capabilities of the `docker container prune`, `docker image prune`, `docker network prune`, and `docker volume prune` commands. You can set filtering options with parameters such as `containers_filters`.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_prune_module.html*.

"docker_image": Building and Managing Images

```
- name: Pull image from Docker Hub
  docker_image:
    name: alpine:latest
    source: pull

- name: Tag image
  docker_image:
    name: alpine:latest
    repository: myalpine:v1.0
    source: local
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

This module is one of the more complex ones from the Docker family because it allows you to build, load, pull, tag, and delete images. With it, you can build images based on Dockerfiles, which is not a particularly Ansible-appropriate task. I will discuss alternatives for building images later.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_image_module.html*.

"docker_image_info": Querying Image Information

```
- name: Query image information
  docker_image_info:
    name: myalpine:v1.0
  register: result

- debug: var=result
```

With the `docker_image_info` module, you can query information about images. It corresponds to the `docker image inspect` command.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_image_info_module.html*.

"docker_compose_v2": Interfacing with Docker Compose

If you know and like Docker Compose, you have the corresponding Ansible interface at hand with the `docker_compose_v2` module. I will give a long example of how to use it in Section 11.4.

Note

There is also the old `docker_compose` module, which works internally with the old `docker-compose` V1. As a Docker user, you'll know that Compose V1 has been obsolete for years and has been replaced by the Compose plug-in (V2).

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/docker/docker_compose_v2_module.html*.

11.3 An Example Application

After all the theory, you must want to see some Docker modules in action—so let's start with a manageable example setup, as shown in Figure 11.1.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

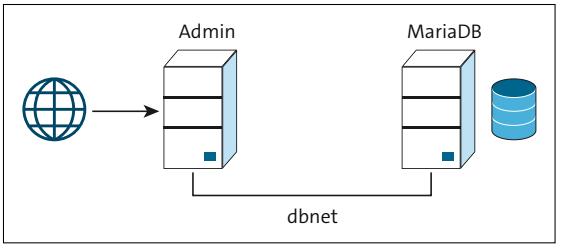


Figure 11.1 Example Setup Consisting of Two Containers

This setup consists of the following components:

- A MariaDB container (depicted by unmodified image directly from Docker Hub)
- A volume for the persistence of MariaDB data
- An Adminer container (also depicted by an unmodified image directly from Docker Hub)
- A port forwarding so that Adminer is accessible from outside
- A private network that connects the two containers

The playbook in Listing 11.2 implements this scenario. Feel free to place it with your other playbooks on the control host:

```
---
- hosts: ubuntu

  vars:
    network_name:      dbnet
    volume_name:       mariadb_data
    db_admin_password: secret
    web_port:          8080

  tasks:
    - name: Create a dedicated network
      docker_network:
        name: "{{network_name}}"

    - name: Create volume for MariaDB data
      docker_volume:
        name: "{{volume_name}}"


    - name: Create MariaDB container
      docker_container:
        name: mariadb
        image: mariadb
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
      networks:
        - name: dbnet
      volumes:
        - "{{volume_name}}:/var/lib/mysql"
      restart_policy: unless-stopped
      env:
        MARIADB_ROOT_PASSWORD: "{{db_admin_password}}"

  - name: Create Adminer container
    docker_container:
      name: adminer
      image: adminer
      networks:
        - name: dbnet
      network_mode: default
      restart_policy: unless-stopped
      env:
        ADMINER_DEFAULT_SERVER: mariadb
      published_ports:
        - '{{web_port}}:8080'
```

Listing 11.2 "mariadb-adminer.yml": Example Setup with Two Docker Containers

If you're familiar with Docker, then these tasks will not be a mystery to you. On the first run, you need to be patient with the two container tasks because the images must first be downloaded from Docker Hub:

```
$ ansible-playbook mariadb-adminer.yml
[…]
```

After that, the application should be accessible on the target machine under the chosen port 8080, as Figure 11.2 shows. If you want to log in, the user is `root` and the password is `secret`.

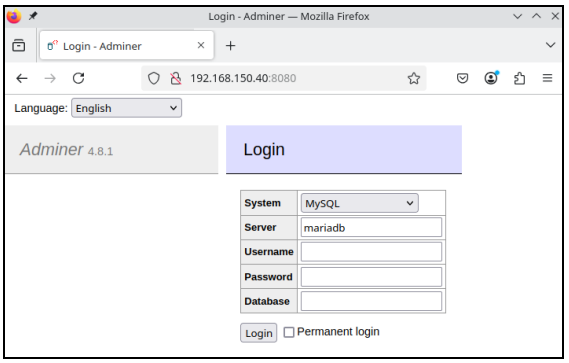


Figure 11.2 Admin Login Page

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

In a test environment, you now need a way to completely remove such a scenario without leaving any traces. Unfortunately, a corresponding playbook for this doesn't just appear out of nowhere, so I've provided this one for you:

```
---
- hosts: ubuntu

  vars:
    network_name:      dbnet
    volume_name:       mariadb_data
    remove_volume:     yes

  tasks:
    - name: Remove containers
      docker_container:
        name: "{{item}}"
        state: absent
      with_items:
        - mariadb
        - adminer

    - name: Remove network
      docker_network:
        name: "{{network_name}}"
        state: absent

    - name: Remove data volume
      docker_volume:
        name: "{{volume_name}}"
        state: absent
      when: remove_volume | bool
```

Listing 11.3 "remove_mariadb-adminer.yml": Stopping and Removing Example Setup

When you start it, the complete *inclusive data volume* setup will be deleted. You can also retain the data if you need to by using the `remove_volume` variable:

```
You use this to delete everything:
$ ansible-playbook remove_mariadb-adminer.yml

You use this to retain data volume:
$ ansible-playbook remove_mariadb-adminer.yml -e remove_volume=no
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

11.4 Ansible and Docker Compose

Anyone who has worked with Docker would probably never manage a multi-container setup like the one from the last section "manually" by using `docker` commands, even without Ansible. The standard tool for this purpose is *Docker Compose*. (We're still talking about single-node systems and not Kubernetes clusters, for example.)

Typically, you'll already have a *docker-compose.yml* file, which for our example might look something like this:

```
---
services:
  mariadb:
    image: mariadb
    environment:
      - MARIADB_ROOT_PASSWORD=secret
    volumes:
      - data:/var/lib/mysql
    restart: unless-stopped

  adminer:
    image: adminer
    environment:
      - ADMINER_DEFAULT_SERVER=mariadb
    ports:
      - 8080:8080
    restart: unless-stopped
    depends_on:
      - mariadb

volumes:
  data:
```

Listing 11.4 "docker-compose.yml": Example Docker Compose File

This description also uses YAML as a language, but it has nothing to do with Ansible! This file is more of a control file for the so-called Docker Compose plug-in. You would typically place it on the Docker host in some project folder (e.g., *~/adm/docker/mariadb-adminer*) so that you could execute various commands for managing the container group there:

```
$ cd ~/adm/docker/mariadb-adminer
(This is where the above file docker-compose.yml is located.)

You use this to start up the container group:
$ docker compose up -d
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
You use this to view logs:
$ docker compose logs

You use this to restart services:
$ docker compose restart

You use this to discard everything (except data volumes):
$ docker compose down
```

These are just some of the many possibilities. In addition, you have conveniences like automatically creating a private network or project-specific prefixes before containers, networks, and volume names. Recreating all this with Ansible would be sheer madness, and you would still not have an adequate replacement for interactive commands like `docker compose exec adminer bash`, which you need in daily life.

This naturally raises the question: Where does Ansible still find its place here if Docker Compose is already an optimal solution?

I think Ansible should retreat to its role as a provisioning tool at this point, so we should use Ansible to ensure that the working environment for Docker Compose is set up. This still gives us many advantages:

- All files are centrally managed on the control host.
- You can implement the *docker-compose.yml* file as a Jinja template, thus achieving dynamic effects that wouldn't be possible with Docker Compose alone.
- During operation, you can manage your services on the Docker host as usual if you need to.

Let me show you an example of how that might look in Listing 11.5. (Of course, you could implement all of this much more elegantly with a role—but for the sake of clarity, I'm only showing a simple Playbook that you can place on your Ansible control host as usual.):

```
---
- hosts: ubuntu

  vars:
    docker_user:       vagrant
    project_name:      mariadb-adminer
    db_admin_password: secret
    web_port:          8080

  tasks:
    - set_fact:
        compose_project_dir: ~{{docker_user}}/adm/docker/{{project_name}}
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    - name: Create project directory
      file:
        path: "{{compose_project_dir}}"
        state: directory
        owner: "{{docker_user}}"

    - name: Upload docker-compose.yml
      template:
        src: docker-compose.yml.j2
        dest: "{{compose_project_dir}}/docker-compose.yml"
        owner: "{{docker_user}}"

    - name: Start containers
      community.docker.docker_compose_v2:
        project_src: "{{compose_project_dir}}"
        state: present
```

Listing 11.5 "compose-mariadb-adminer.yml": Ansible and Compose in Interaction

You can easily reuse this file by extracting the Compose project name as a variable. The last task is the invocation of the `docker_compose_v2` module, which assumes that the *docker-compose.yml* file is already present on the target host. You can check or transfer this in the penultimate task by using the `template` module.

Next, you place the required template file as usual next to the playbook or in a *templates/* subfolder:

```
---
services:
  mariadb:
    image: mariadb
    environment:
      - MARIADB_ROOT_PASSWORD={{db_admin_password}}
    volumes:
      - data:/var/lib/mysql
    restart: unless-stopped

  adminer:
    image: adminer
    environment:
      - ADMINER_DEFAULT_SERVER=mariadb
    ports:
      - {{web_port}}:8080
    restart: unless-stopped
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    depends_on:
      - mariadb

volumes:
  data:
```

Listing 11.6 "docker-compose.yml.j2": Underlying Compose File as Jinja Template

We've used the Jinja features very sparingly here; only the forwarding port and the admin password are parameters. But with Jinja, you could, of course, do *anything*.

11.4.1 Version Issues

Now, at least in our lab, there is still a small problem: the `docker_compose_v2` module was only added in *community.docker* version 3.6.0, and unfortunately, our control host cannot provide that:

```
$ ansible-galaxy collection list | grep community.docker
community.docker             3.4.7
```

In the lab, you can quickly solve this by installing the latest version:

```
$ ansible-galaxy collection install community.docker
```

Now, everything is ready to roll out our small example setup:

```
$ ansible-playbook compose-mariadb-adminer.yml
```

You can now remove the setup on the Docker host with `docker compose down` or with an Ansible ad-hoc command from the control host:

```
$ ansible ubuntu -m community.docker.docker_compose_v2 \
    -a "project_src=~vagrant/adm/docker/mariadb-adminer state=absent"
```

11.4.2 Inline Specification with "definition"

In conclusion, I'd like to mention that when calling the `docker_compose_v2` module, you could alternatively specify the complete Docker Compose definition inline (i.e., in the playbook) by using the `definition` parameter instead of the approach with `project_src` that I've shown you. (See the examples in the official documentation.) However, I don't see much benefit in doing that—you'll have less flexibility than if you used an external template file, and you'll have *nothing* on the Docker host if you ever want to do something manually with `docker compose` there.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

11.5 The "docker" Connection Plug-in

In the container universe, tinkering with running containers is generally considered an anti-pattern—but always following the rules isn't fun, either. Besides, you might just want to copy a file from a container or fetch some other information. If you want to do this with Ansible on top of everything else, you'll initially face the problem that there is never an SSH server inside a container. (That would indeed be a real anti-pattern!) But Ansible wouldn't be Ansible if there weren't other options.

To help you execute commands inside containers, the `community.docker` collection has provided the aforementioned `docker_container_exec` module since version 1.5.0. (It's not available in older Ansible community versions.)

Another option is the `docker` Connection plug-in, but it's only straightforward to use if Ansible and Docker are running on the same machine. In our case, this is true on the Ubuntu system (where we have installed Ansible, making it the control host). As an example, you can start a minimal Python container there:

```
$ docker run -itd --name py1 python:3-slim
```

Why should you start a *Python* container? It's because Ansible only works well if Python is present on the target system (target container). With the `docker` Connection plug-in, you can now address and edit running containers that are equipped accordingly by their name:

```
$ ansible all -c docker -i py1, -m ping
py1 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/local/bin/python3.12"
    },
    "changed": false,
    "ping": "pong"
}
```

If a target container does not have Python on board, then only a few modules like `raw` and `script` will work. Therefore, you won't be able to use modules, like `copy` and `fetch`, that you need to copy files. For that reason, you'll have to manage with a `docker cp` wrapped in a `command`. (Which means you won't need a Connection plug-in.)

If the `docker` Connection plug-in is available for your task, and if you need a playbook, then you can set the `connection` parameter at the play level instead of on the command line:

```
---
- hosts: all
  connection: docker
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
  tasks:
    - ping:
```

Listing 11.7 "docker-connection.yml"

Since we don't have a suitable inventory in this quick example, our temporary inventory will remain part of the command line:

```
$ ansible-playbook -i py1, docker-connection.yml
```

11.6 Creating Images

When you want a tool to help you create custom Docker images, the Docker tool set often leaves much to be desired. Anyone who has written complex Dockerfiles will agree that it is not the most enjoyable of tasks. Even once it is completed, Dockerfiles are far from clear and easy to maintain. (While the new Docker Buildx plug-in introduces many useful features, it doesn't fundamentally change the way images are created in Docker.)

A Dockerfile essentially describes how to create a new image by incrementally adding features to a base image. First, I'll show you an example of what this typically looks like in the Docker world, and then, we'll discuss a way in which Ansible can (perhaps) help here.

11.6.1 Creating Images with "docker build"

To demonstrate the methods of image creation, we'll build an example `cgiapp` image that's based on the official `httpd` image and that contains a minimalist computer-generated imagery (CGI) web application written in Python. First, we'll choose the classic way and use only the Docker tool set.

First, you go to a Docker host and start by creating a project folder, like this one:

```
$ mkdir -p ~/docker/build/cgiapp && cd $_
```

Then, you put the following two files in the following folder.

First, you add the source code of the application:

```
#!/usr/bin/env python3

from flask import Flask
from wsgiref.handlers import CGIHandler

app = Flask(__name__)
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
@app.route('/')
def hello_world():
    return "<h1>Hello, World!</h1>"

if __name__ == '__main__':
    CGIHandler().run(app)
```

Listing 11.8 "app.py": Python/Flask-based Hello World CGI

And second, you add the Dockerfile (which is the control file for the build process):

```
FROM httpd:2.4

RUN apt update && apt install -y python3-flask

WORKDIR /usr/local/apache2

COPY app.py ./cgi-bin/
RUN chmod 755 ./cgi-bin/app.py

RUN sed -ri 's/#(LoadModule cgid.*)/\1/'     ./conf/httpd.conf
RUN sed -ri 's/#(LoadModule rewrite.*)/\1/'  ./conf/httpd.conf

RUN echo "RewriteEngine On"                          >>./conf/httpd.conf
RUN echo "RewriteRule ^/$ /cgi-bin/app.py [NC,PT]" >>./conf/httpd.conf
```

Listing 11.9 "Dockerfile": Control File for Building Image

The exciting second half of the Dockerfile activates two required modules in the Apache server. (The `LoadModule` lines are commented out by default.) Then, an internal redirection is defined so that the application is accessible not only with the complicated `/cgi-bin/app.py` URL but also simply with `/`.

Next, you build the image and start a container with it:

```
Note the period at the end!
$ docker build -t localhost/cgiapp:1.0 .

$ docker run -d -p 8081:80 --name cgiapp localhost/cgiapp:1.0
```

If everything goes well, you can now access the application using a browser or a tool like `curl` at the specified forwarding port when the container was started:

```
$ curl localhost:8081
<h1>Hello, World!</h1>
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

You can then discard the container with this:

```
$ docker rm -f cgiapp
```

All of this obviously has nothing to do with Ansible; as mentioned, it's only meant to demonstrate how image creation in the Docker world usually looks without any additional tools.

11.6.2 "ansible-bender"

Instead of using a Dockerfile to execute build steps, you could take a Docker base image (or more precisely, a base container) and run an Ansible playbook inside it. That way, rather than being limited by Dockerfile syntax, you could gain access to all the powerful capabilities of Ansible. Reproducibility would be built in, and with the Docker connection plug-in we just discussed, access becomes highly efficient.

A few years ago, this clever idea led to the creation of the `ansible-container` project, which showed great promise even though it needed refinement (e.g., caching). However, by 2019, the project was discontinued and marked as deprecated. Why? It was likely because the project tried to achieve too much from a technical standpoint and insurmountable challenges therefore arose. Yet, a successor quickly emerged: `ansible-bender` (see *https://ansible-community.github.io/ansible-bender/build/html/*).

Now, let's take a closer look at this tool.

Currently, the `ansible-bender` is usable only with Podman and Buildah and not with Docker. However, this is not a real problem because images created with Buildah are fully compatible with Docker. You just need to install these tools to get started, and the installation does not conflict with Docker, so you can install them on the same host. You do it as follows:

1. Install `ansible-bender`:

   ```
   $ sudo pip3 install ansible-bender --break-system-packages
   ```

2. Podman and Buildah will then be available in all distributions:

   ```
   $ sudo apt install podman buildah
   ```

You can simply think of Podman as a drop-in replacement for Docker. For example, try this:

```
$ podman run --rm hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.
[...]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

This works just as well as an analog `docker` call. (Behind the scenes, Podman behaves slightly differently than Docker—the keyword is *server-less*—but that's not what we're interested in here). For us, all of this is just a means to the actual end: creating Docker images with Ansible or `ansible-bender`, which we can now come to grips with.

11.6.3 Creating Images with "ansible-bender"

You should start with a new project folder:

```
$ mkdir -p ~/ansible/bender/cgiapp && cd $_
```

Before we can fully utilize `ansible-bender`, there's a minor obstacle to overcome: the base image intended for use with `ansible-bender` must include Python, as this is a requirement for Ansible—and of course, the standard `httpd` image from Docker Hub doesn't have Python on board.

Firstly, you must construct your own base image that fulfills all of the necessary criteria. You can proceed with this step "traditionally" by using a Dockerfile:

```
FROM docker.io/httpd:2.4
RUN  apt update && apt -y install python3 python3-apt
```

Listing 11.10 "Dockerfile"

The, you can quickly create the new base image with `buildah`:

```
$ buildah bud -t localhost/myhttpd:2.4 .
```

(The mysterious abbreviation `bud` means "build using Dockerfile.")

Now, you can finally proceed to building your application image. The playbook from Listing 11.11 will be the control file for `ansible-bender`. (You can choose a different filename if you like.):

```
---
- name: ansible-bender Playbook
  hosts: all
  vars:
    ansible_bender:
      base_image: "localhost/myhttpd:2.4"
      target_image:
        name: localhost/cgiapp:2.0

  tasks:
    - name: Install python3-flask
      apt:
        update_cache: yes
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
      name: python3-flask

  - name: Activate modules cgid and rewrite in httpd.conf
    replace:
      path: /usr/local/apache2/conf/httpd.conf
      regexp: '#(LoadModule {{item}}.*)'
      replace: '\1'
    with_items: [cgid, rewrite]

  - name: Copy app
    copy:
      src: app.py
      dest: /usr/local/apache2/cgi-bin
      mode: 0755

  - name: Configure redirect
    blockinfile:
      path: /usr/local/apache2/conf/httpd.conf
      block: |
        RewriteEngine On
        RewriteRule ^/$ /cgi-bin/app.py [NC,PT]
```

Listing 11.11 "build.yml": Image Creation with "ansible-bender"

You can place the *app.py* file from Section 11.6.1 in the same folder, too. (The application hasn't changed due to the new build process.)

You can now initiate image creation as follows:

```
$ ansible-bender build build.yml
[…]
Image 'cgiapp' was built successfully \o/
```

If everything goes well, you can also start a container from the newly created image with Podman:

```
$ podman run -d -p 8081:80 --name cgiapp localhost/cgiapp:2.0
```

If that works as well, you can access the application in your browser or with a tool like `curl` by using the forwarding port you specified when you started the container:

```
$ curl localhost:8081
<h1>Hello, World!<h1>
```

And when you are done trying it out, you can discard the container with this:

```
$ podman rm -f cgiapp
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Importing Images from Podman into Docker

We actually wanted to build Docker images, but `ansible-bender` has now provided us with a Podman image. Due to the compatible image formats, it's just a tiny step to import an image from the Podman world into the Docker world:

```
$ podman save cgiapp:2.0 | docker load
```

11.6.4 Conclusion

The excitement of being able to build Docker images with Ansible is tempered by the requirement of having a Python interpreter in the base image. While this might not be an issue for everyone, ideally, containers should be kept as minimal as possible. As a compromise, you could remove Python in the final task; however, due to the architecture of Docker images, that doesn't actually reduce the final size.

Furthermore, I would like to point out that if you encounter serious problems during the build process and need to ask someone, you'll probably find that *few people* are familiar with this build method! Everyone knows Dockerfiles, and the internet is full of solutions to all kinds of problems—but with `ansible-bender`, you'll likely be quite alone if you have a question.

In short, I'm not trying to dissuade you from using the Ansible tool. Give it a try if you believe the technique has potential or if, after working with traditional Dockerfiles, you think, "There must be a better solution out there." Personally, as an experienced Docker user, I'll stick with Docker for now, despite my fondness for Ansible. Alternatively, I might explore the Podman/Buildah ecosystem, as there are options beyond just Dockerfiles.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 12
Inventory Management: Advanced Methods

In Chapter 2, Section 2.3, we looked at simple static inventories and intentionally left open some points on inventory management. Now is a good time for us to revisit inventories.

12.1 The "ansible-inventory" Command

Let's start with another useful tool from the Ansible suite: `ansible-inventory`. This command allows you to gather information about your inventory and display the configuration of hosts.

Try the following calls:

```
Use this to get an overview (ASCII graphic), without and with group parameterization:
ansible-inventory --graph
ansible-inventory --graph --vars

This is the same call, restricted to a group:
ansible-inventory --graph <GROUPNAME>
ansible-inventory --graph --vars <GROUPNAME>

Use this to get an overview (JSON or YAML) with parameterization:
ansible-inventory --list
ansible-inventory --list --yaml

Use this to get an overview of the parameterization (JSON or YAML) of individual hosts:
ansible-inventory --host <HOSTNAME>
ansible-inventory --host <HOSTNAME> --yaml
```

Here, you can see an example output:

```
$ ansible-inventory --graph
@all:
  |--@test_hosts:
  |  |--debian
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
  |  |--rocky
  |  |--suse
  |  |--ubuntu
  |--@ungrouped:
```

12.2 Nested Groups

Unlike user groups in Linux, groups in an Ansible inventory can be nested, meaning you can define groups of groups. You define such a "supergroup" in the static inventory by appending the `:children` suffix to the group name:

```
[debian_hosts]
debian

[rocky_hosts]
rocky

[suse_hosts]
suse

[ubuntu_hosts]
ubuntu

[apt_hosts:children]
debian_hosts
ubuntu_hosts

[rpm_hosts:children]
rocky_hosts
suse_hosts
```

Listing 12.1 "inventory": Example of Nested Groups

In this example, you see the two `apt_hosts` and `rpm_hosts` groups, each of which contains the members of the listed simple groups. Of course, you can parameterize supergroups with `:vars`, just as you are already familiar with from normal groups.

Let's test if this also works:

```
$ ansible-inventory --graph apt_hosts
@apt_hosts:
  |--@debian_hosts:
  |  |--debian
  |--@ubuntu_hosts:
  |  |--ubuntu
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

By the way, the nesting depth is not limited; if it's useful for you, you can define groups of groups of groups, etc.

12.3 Static Inventories in YAML format

The well-known "INI-like" inventory format is not the only possible one in Ansible. Alternatively, you can write your static inventory files in YAML syntax. Before we delve into the question of *why* you might want to do that, let's take a look at such a YAML inventory.

Starting from scratch, creating a YAML inventory would be considerably more challenging than using the traditional format. But we already have an inventory, and with the `ansible-inventory --list --yaml` command, you basically get the content you need. You just need to add the `--export` switch to exclude the facts from the facts cache and consolidate it a bit:

```
$ ansible-inventory --list --yaml --export
all:
  children:
    test_hosts:
      hosts:
        debian:
          ansible_host: 192.168.150.10
        rocky:
          ansible_host: 192.168.150.20
        suse:
          ansible_host: 192.168.150.30
        ubuntu:
          ansible_host: 192.168.150.40
      vars:
        ansible_become: 'yes'
        ansible_user: vagrant
    ungrouped: {}
  vars:
    color: LightGray
    number: 50
```

The `vars` section in this example is from a *group_vars/* file, and the variables do not need to be repeated in the inventory. We also don't need an empty `ungrouped` map, and with a bit of manual editing, you can quickly adjust this output:

```
$ ansible-inventory --list --yaml --export >inventory.yml

Then, you edit with your favorite editor.
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

In the end, you'll have something like this:

```
---
all:
  children:
    test_hosts:
      hosts:
        debian:
          ansible_host: 192.168.150.10
        rocky:
          ansible_host: 192.168.150.20
        suse:
          ansible_host: 192.168.150.30
        ubuntu:
          ansible_host: 192.168.150.40
      vars:
        ansible_become: 'yes'
        ansible_user: vagrant
```

Listing 12.2 "inventory.yml": Simple Example of YAML-Based Inventory

Initially, this takes some getting used to, but the structure is ultimately logical. If you had several layers of nested groups, you would represent that in YAML with much deeper levels of indentation or nesting.

But first, you should check whether the new inventory is functional and behaves identically to the old inventory:

```
$ ansible all -i inventory.yml -m ping
debian | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
[…]
```

The question of why and what for remains. Simplicity can hardly be the reason—although "simple" is always relative. If you wanted to automatically create the inventory through code, then YAML would indeed be simpler as the target format, as all common programming languages come with powerful YAML libraries.

A compelling reason to use a new inventory is because you need to store structured variables in it, which you can do seamlessly with a YAML inventory:

```
      vars:
        # ...
        animals:
          - name: Lassie
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
      age:  7
    - name: Kitty
      age:  9
```

Listing 12.3 “inventory.yml”: YAML Inventory Excerpt with Structured Variables

You can also do this in an INI inventory, but you would have to take the somewhat more painful route via JSON:

```
[test_hosts:vars]
# ...
animals=[ {"name": "Lassie", "age": 7}, {"name": "Kitty", "age": 9} ]
```

Listing 12.4 “inventory”: INI Inventory Excerpt with Structured Variables

However, this also works smoothly once you’ve specified the data structure in the inventory without errors:

```
$ ansible debian -m debug -a "msg={{ animals[1]['name'] }}"
debian | SUCCESS => {
    "msg": "Kitty"
}
```

My conclusion is that an INI inventory allows you to handle all practical tasks effectively. However, having YAML as an alternative is beneficial because it provides users with the flexibility to choose the option that best fits their needs.

12.4 Creating on-the-Fly Inventories with “add_host”

Sometimes, you may want to use Ansible to manage target hosts that are not listed in an inventory—for example, because the systems have just been provisioned and adequate dynamic inventory handling isn’t available.

In such situations, the `add_host` module can be quite useful because it allows you to add hosts and groups to a temporary inventory for use in subsequent plays. This inventory exists only in memory and is discarded after playbook execution.

In the following example, the task in the first play places two of our already existing hosts under a new name into a group called `temp_group`. The second play then uses these hosts:

```
---
- hosts: localhost
  tasks:
    - name: Add hosts
      add_host:
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
        name: "{{item.name}}"
        groups: temp_group
        ansible_host: "{{item.ip}}"
        ansible_user: vagrant
        ansible_become: yes
      changed_when: false
      with_items:
        - name: srv1
          ip:   192.168.150.10
        - name: srv2
          ip:   192.168.150.20

- hosts: temp_group
  tasks:
    - name: Say hello
      debug: msg="Hello {{inventory_hostname}}"

    - name: Proof that become is working
      command: test -r /etc/shadow
```

Listing 12.5 "add_host.yml": Example Use of "add_host"

Whether you should give the `add_host` task a `changed_when: false` status is probably a matter of personal preference. In my opinion, one factor *in favor* of doing it is that setting up a temporary Ansible group does not cause any actual changes to the `localhost` system.

To prove beyond doubt that our existing static inventory does not affect the environment's behavior, we can call the playbook with an empty inventory:

```
$ ansible-playbook -i, add_host.yml
[WARNING]: provided hosts list is empty, only localhost is available.
Note that the implicit localhost does not match 'all'

PLAY [localhost] ***************************************************************

TASK [Add hosts] ***************************************************************
ok: [localhost] => (item={'name': 'srv1', 'ip': '192.168.150.10'})
ok: [localhost] => (item={'name': 'srv2', 'ip': '192.168.150.20'})

PLAY [temp_group] **************************************************************

TASK [Gathering Facts] *********************************************************
ok: [srv2]
ok: [srv1]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
TASK [Say hello] ***********************************************************
ok: [srv1] => {
    "msg": "Hello srv1"
}
ok: [srv2] => {
    "msg": "Hello srv2"
}

TASK [Proof that become is working] ****************************************
changed: [srv1]
changed: [srv2]

PLAY RECAP *****************************************************************
localhost                  : ok=1    changed=1    unreachable=0    failed=0
srv1                       : ok=3    changed=1    unreachable=0    failed=0
srv2                       : ok=3    changed=1    unreachable=0    failed=0
```

12.5 Dynamic Groups with “group_by”

Another very powerful feature of Ansible is the `group_by` module, which allows you to create dynamic groups during a playbook run. What is it used for? Imagine you have organized your inventory by locations or topics (mail servers, web servers, etc.), but now, you need to group them by operating systems (Debian, SUSE, etc.).

If you only need this new grouping to be available dynamically and you don’t need to permanently record it in the inventory, then you can easily achieve your goal with `group_by`. Essentially, you can use any facts to create a group classification.

To see how, in the first play, a grouping by operating system families is formed and how, in further plays, the hosts are processed group by group, see Listing 12.6:

```
---
- name: Group by OS family
  hosts: all

  tasks:
    - group_by: key=os_{{ ansible_os_family }}


- name: Process all members of the Debian family
  hosts: os_Debian

  tasks:
    - debug: msg="Hello {{ inventory_hostname }}"
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
- name: And now the rest
  hosts: all:!os_Debian

  tasks:
    - debug: msg="Hello {{ inventory_hostname }}"
```

Listing 12.6 "group_by.yml": Dynamic Grouping by Operating System Families

Here's what the process looks like:

```
$ ansible-playbook group_by.yml
[…]
PLAY [Process all members of the Debian family] ****************************

TASK [debug] ***************************************************************
ok: [debian] => {
    "msg": "Hello debian"
}
ok: [ubuntu] => {
    "msg": "Hello ubuntu"
}

PLAY [And now the rest] ****************************************************

TASK [debug] ***************************************************************
ok: [rocky] => {
    "msg": "Hello rocky"
}
ok: [suse] => {
    "msg": "Hello suse"
}
[…]
```

Of course, you don't have to use all the groups of hosts that have been created. (There would have been `os_Suse` and `os_RedHat`.) A `group_by` only provides groups, and what you do with them is entirely up to you.

Listing 12.7 shows another idea: grouping by the installed version of a software. (We use Python as an example here.) Another interesting aspect is the multi-line Jinja `if` statement, which becomes clearer when formatted like this:

```
---
- hosts: all

  tasks:
    - name: Install Python rpm module on SUSE
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
      zypper:
        name: python3-rpm
      when: ansible_os_family == "Suse"

    - name: Retrieve package information
      package_facts:

    - name: Determine version of installed python3
      set_fact:
        python_version: >-
          {{ ansible_facts.packages['python3'][0]['version'] }}


    - group_by:
        key: |-
          {% if python_version is version('3.10', '>=') %}
          {{ "python_new" -}}
          {% else %}
          {{ "python_old" -}}
          {% endif %}


- hosts: python_new

  tasks:
    - debug: msg="Python version {{python_version}}"

- hosts: python_old

  tasks:
    - debug: msg="Python version {{python_version}}"
```

Listing 12.7 “group_by_pythonversion.yml”: Dynamic Grouping by Installed Python Version

This will give you a general idea of its appearance:

```
$ ansible-playbook group_by_pythonversion.yml
[…]
PLAY [python_new] ************************************************************

TASK [debug] *****************************************************************
ok: [debian] => {
    "msg": "Python version 3.11.2-1+b1"
}
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
ok: [ubuntu] => {
    "msg": "Python version 3.12.3-0ubuntu2"
}

PLAY [python_old] ************************************************************

TASK [debug] *****************************************************************
ok: [rocky] => {
    "msg": "Python version 3.9.18"
}
ok: [suse] => {
    "msg": "Python version 3.6.15"
}
[…]
```

12.6 Dynamic and External Inventories

Our well-known static inventory sometimes reaches its limits. For example, if the number of target hosts fluctuates greatly and/or the target hosts are already inventoried elsewhere, then maintaining an additional static Ansible inventory can become very labor intensive and error-prone. Ansible's ability to integrate *dynamic* and *external inventories* provides a remedy.

There are basically two methods:

1. Inventory scripts (the classic method)
2. Inventory plug-ins (the newer method)

With both methods, you can either rely on existing solutions or develop new ones yourself (meaning you can code them). The key difference is that you can create an inventory script in any programming language, whereas you can only write an inventory plug-in in Python.

For practical reasons, we'll primarily focus on method 2 in Chapter 13 because it pertains to cloud environments where dynamic inventories quickly become a relevant topic.

12.6.1 Inventory Scripts

Instead of a static INI or YAML file, you can provide Ansible with an *executable program* as an inventory source. Whenever inventory information is needed, Ansible will start this program and process its output as inventory. However, the Ansible developers have introduced a minor challenge: the only accepted output format is JSON. To get a

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

quick impression of the required syntax, you need to export your existing classic inventory with the usual command:

```
$ ansible-inventory --list --export
```

This gives you a good impression of what your own inventory script would need to produce.

Let's explore a playful example to demonstrate the fundamental mechanisms. Suppose you manage your hosts in a comma-separated values (CSV) file like this:

```
host,ip,group,user,become
debian,192.168.150.10,test_hosts,vagrant,yes
rocky,192.168.150.20,test_hosts,vagrant,yes
suse,192.168.150.30,test_hosts,vagrant,yes
ubuntu,192.168.150.40,test_hosts,vagrant,yes
```

Listing 12.8 "hosts.csv": Simple CSV File with Inventory Information

(The format is quite rigid, but the example is intended to be as straightforward as possible.)

From this, you now need to generate a JSON output that is accepted by Ansible. (You can find more details at *https://docs.ansible.com/ansible/latest/dev_guide/developing_inventory.html#developing-inventory-scripts*.)

The script in Listing 12.9 is an example of how you could accomplish this using the Python programming language:

```
#!/usr/bin/env python3

import csv
import json
import sys
from collections import defaultdict

CSV_FILE = 'hosts.csv'

def main():
    inventory = defaultdict(lambda: {"hosts": [], "vars": {}})
    hostvars = {}

    # Parse CSV data:
    with open(CSV_FILE, newline='') as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            host   = row["host"]
            ip     = row["ip"]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
            group  = row["group"]
            user   = row["user"]
            become = row["become"]

            # Add host to group:
            inventory[group]["hosts"].append(host)

            # Save ip, user, become as host vars:
            hostvars[host] = {
                "ansible_host": ip,
                "ansible_user": user,
                "ansible_become": become
            }

    # Add hostvars as _meta:
    inventory["_meta"] = {"hostvars": hostvars}

    # Called with --list ?
    if len(sys.argv) == 2 and sys.argv[1] == '--list':
        print(json.dumps(inventory, indent=2))
    else:
        print(json.dumps({}))


if __name__ == '__main__':
    main()
```

Listing 12.9 "inv_from_csv.py": Example Inventory Script in Python

Next, you place the two files side by side in the project folder. Remember to mark the script as executable:

```
$ chmod +x inv_from_csv.py
```

Now, you can start and test it entirely independently of Ansible:

```
$ ./inv_from_csv.py --list
{
  "test_hosts": {
    "hosts": [
      "debian",
      "rocky",
      "suse",
      "ubuntu"
    ],
[…]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The key is that the script functions as intended within the Ansible framework. You simply use an inventory script by designating it as an inventory source (e.g., with the `-i` switch). Try it:

```
$ ansible all -i ./inv_from_csv.py -m ping
[…]

$ ansible all -i ./inv_from_csv.py -a "test -r /etc/shadow"
[…]
```

12.6.2 Using Inventory Plug-ins

When you specify an inventory (e.g., with the `-i` switch), you also have the option to specify the configuration file of an *inventory plug-in*. This essentially implements the inventory script variant we mentioned previously, and it does it internally, solely via a plug-in.

The configuration file must be in Ansible's typical YAML format, and the specific syntax is described in the documentation of each respective plug-in. The general, official documentation on the topic is at *https://docs.ansible.com/ansible/latest/plugins/inventory.html*, and you can obtain an overview of all the inventory plug-ins that are available on your system by using the following command:

```
$ ansible-doc -t inventory -l
```

As mentioned earlier, I direct your attention to Chapter 13, where we will explore various cloud environments and their corresponding inventory plug-ins in detail.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 13
Ansible and the Cloud

Cloud computing has been one of the most important topics in the IT industry for almost two decades. The term *cloud* encompasses a vast concept, and it's often necessary to first agree on what exactly it means to you. My personal (pragmatic) favorite definition is this:

cloud (n.): other people's computers

Essentially, when I use the cloud, I'm paying others to access and utilize their computers.

We want to examine the topic primarily from an Ansible perspective. Of particular interest is the infrastructure as a service (IaaS) layer, which involves the management of virtual machines in the cloud (again, this is very simplified). And because Ansible doesn't really care *where* the machines are located once they're available, in this chapter, we'll mostly look at the provisioning and inventorying of cloud servers. You likely have a good idea of what comes next.

Attention: It's About Your Money!

A common feature of all public cloud environments is that cloud servers cost money as long as they *exist*. It doesn't matter whether they are on or off, so remember to delete test servers in the cloud when you no longer need them.

Our experiments—if you want to follow them live—probably won't break your bank. For example, with the German provider Hetzner (one of the internationally operating larger German providers), the smallest server currently costs €4.51 per month, and that includes an IPv4 address with billing by the hour. Nevertheless, you should always keep a close eye on the prices and conditions of your specific provider!

I would like to demonstrate Ansible's cloud capabilities using two public cloud environments as examples: the environment of Hetzner and that of the top dog: Amazon. If you're more interested in Microsoft Azure, Digital Ocean, Google Cloud, or VMware, you won't find specific information on it here. However, the concepts are quite similar, so taking a quick look at Hetzner or Amazon could still be beneficial for you.

After that, we'll explore the increasingly popular *Proxmox VE* private cloud, which you can also manage with Ansible.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

13.1 Version Issues and virtualenv

Currently, we're running an `ansible-core` version 2.14.x on our lab's control host. However, in the cloud environment, we generally recommend that you to use the most current versions of the respective collections. This presents a problem because these currently often require `ansible-core` version 2.15.x or later. With a bit of luck, everything might work if you use version 2.14, and you'll often be able to ignore the warning messages—but you'll always have some lingering uncertainty.

For simplicity's sake, in this section, I'll assume that you're using the most recent core version that's available. As outlined in Chapter 1, Section 1.6, we'll utilize a virtualenv environment. You can set this up quickly as follows:

```
$ python3 -m venv ~/venv/ansible-new
$ source ~/venv/ansible-new/bin/activate

$ pip3 install ansible-core

$ ansible --version
ansible [core 2.17.5]
```

Also, you should install the `community.general` collection right away because it contains many useful things that you won't want to miss (e.g., the YAML output plug-in):

```
$ ansible-galaxy collection install community.general
```

13.2 Where to Store Keys, Tokens, Secrets, Etc.

To gain API access to cloud environments, you almost always need some keys, tokens, or similar credentials to prove that you are authorized. Such sensitive data must be stored securely somewhere, and preferably, it'll be encrypted.

If you're considering using Ansible Vault, it's generally a good idea to do so, but it can be somewhat impractical in a cloud environment because it limits the tool set exclusively to Ansible.

In such situations, I recommend that you use a password safe. For simplicity's sake in this chapter, we'll work with `pass`, which you may have already encountered in the section on Ansible Vault.

So, if you need to, you can take a short trip to Appendix B to have a functional password manager available afterward!

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

13.3 Hetzner Cloud

The German company Hetzner provides an excellent and flexible cloud solution. If you want to follow the upcoming topics live, you need to register with Hetzner or have the appropriate access data at hand to log in at *https://console.hetzner.cloud* and perform administrative tasks there.

13.3.1 Preparations on the Control Host

First, you need to perform the following preparatory steps on the control host:

1. *Please* start a new Ansible project to keep it distinct from our past efforts. For ease of use, refer to my script from Chapter 19, Section 19.1.2:

   ```
   $ init-ansible-project.sh ~/ansible/projects/hcloud && cd $_
   ```

2. Then, generate a dedicated *hetzner*|*hetzner.pub* SSH key pair for managing the machines in the Hetzner cloud. The storage location should be outside the project in the usual *~/.ssh/* folder, and you can initially skip a passphrase in the lab:

   ```
   $ ssh-keygen -f ~/.ssh/hetzner -N ""
   Generating public/private rsa key pair.
   […]
   ```

3. Install the *hcloud* Python module, which is required by the Ansible modules we will be using soon. You can easily do this in our virtualenv environment:

   ```
   $ pip3 install hcloud
   ```

4. The most important thing is installing the `hetzner.hcloud` collection:

   ```
   $ ansible-galaxy collection install hetzner.hcloud
   ```

5. Install the `hcloud` command-line client (which is not necessary for Ansible but is still highly recommended). You can get this "fresh" from the internet at *https://github.com/hetznercloud/cli/releases*. The client is Go-based, so you'll have no dependencies on Python. You can also adjust the version number if necessary:

   ```
   $ VERSION=v1.47.0

   $ curl -fsSLO https://github.com/hetznercloud/cli/releases/\
   download/$VERSION/hcloud-linux-amd64.tar.gz

   $ tar -xzvf hcloud-linux-amd64.tar.gz  hcloud

   $ sudo install hcloud /usr/local/bin

   Clean up:
   $ rm hcloud-*
   ```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

13.3.2 Preparations in the Cloud

Assuming you are now logged in to the Hetzner cloud console in your browser, here's what you do next:

1. Create a new project in the cloud console (e.g., "Ansible-Test").
2. Select the new project and switch to the **Security** menu ("Sicherheit").
3. Generate a token in the **API-TOKENS** tab. Clients possessing this token will then be authorized to access API functions of this project. You must use `Ansible` as description text, and for **Permissions** ("Berechtigungen"), you should be sure to select `Read&Write` ("Lesen&Schreiben").

   You should store the token immediately because you won't be able to view later. Place it in the password store like this:

   ```
   $ pass insert ansible/hcloud/api_token
   ```

The `hcloud` command-line tool uses the token from the `HCLOUD_TOKEN` environment variable. You should test it right away, as follows:

```
$ export HCLOUD_TOKEN=$(pass ansible/hcloud/api_token)

$ hcloud server list
ID   NAME   STATUS   IPV4   IPV6   PRIVATE NET   DATACENTER   AGE
```

Of course, in a new project, there are no servers yet, but if something goes wrong with the authentication, you'll see something like this:

```
hcloud: unable to authenticate (unauthorized)
```

In that case, you should carefully check all the steps again or repeat them if necessary.

13.3.3 Using Cloud Modules

Let's begin with a straightforward example: collecting information like an overview of available server types and available images. The following playbook achieves this:

```
---
- hosts: localhost

  tasks:
    - name: Gather hcloud server type infos
      hetzner.hcloud.server_type_info:
      register: info

    - debug: var=info
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    - name: Gather hcloud image infos
      hetzner.hcloud.image_info:
      register: info

    - debug: var=info
```

Listing 13.1 "info.yml": Gathering Information About Server Types

However, this variant assumes that the API token is stored in the `HCLOUD_TOKEN` environment variable:

```
$ export HCLOUD_TOKEN=$(pass ansible/hcloud/api_token)

$ ansible-playbook info.yml
```

The second variant without an environment variable provides the `api_token` attribute to each corresponding task. Conveniently, there is a lookup plug-in in the `community.general` collection for our password manager:

```
---
- hosts: localhost

  vars:
    api_token: "{{ lookup('passwordstore', 'ansible/hcloud/api_token') }}"

  tasks:
    - name: Gather hcloud server type infos
      hetzner.hcloud.server_type_info:
        api_token: "{{ api_token }}"
      register: info

    - debug: var=info

    - name: Gather hcloud image infos
      hetzner.hcloud.image_info:
        api_token: "{{ api_token }}"
      register: info

    - debug: var=info
```

Listing 13.2 "info2.yml": Gathering Information About Server Types

While it may not seem worthwhile in this brief example, I'd still like to demonstrate a third option in which you can keep the actual tasks "clean" by using `module_defaults` (see Chapter 19, Section 19.4.3):

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts: localhost

  vars:
    api_token: "{{ lookup('passwordstore', 'ansible/hcloud/api_token') }}"

  module_defaults:
    hetzner.hcloud.server_type_info:
      api_token: "{{ api_token }}"
    hetzner.hcloud.image_info:
      api_token: "{{ api_token }}"

  tasks:
    - name: Gather hcloud server type infos
      hetzner.hcloud.server_type_info:
      register: info

    - debug: var=info

    - name: Gather hcloud image infos
      hetzner.hcloud.image_info:
      register: info

    - debug: var=info
```

Listing 13.3 "info3.yml": Gathering Information About Server Types

13.3.4 Provisioning Cloud Servers

We'll now assume that the `HCLOUD_TOKEN` environment variable is set correctly.

With the following playbook, you create an initial *small* server in the Hetzner cloud:

```
---
- hosts: localhost

  tasks:
    - name: Upload SSH-key
      hetzner.hcloud.ssh_key:
        name: sshkey_hetzner
        public_key: "{{ lookup('file', '~/.ssh/hetzner.pub') }}"
        state: present

    - name: Create Debian 12 server
      hetzner.hcloud.server:
        name: srv01
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    server_type: cx22
    location: nbg1
    image: debian-12
    ssh_keys:
      - sshkey_hetzner
    labels:
      type: test
      purpose: learning
    state: present
```

Listing 13.4 "provisioning.yml": Provisioning Cloud Server

Uploading the SSH key for this project is a one-time action; it could also take place in another Playbook, or you could do it manually beforehand.

Labels are technically unnecessary, but in practice, they can greatly contribute to the clarity of the project later if you want to sort or group your servers according to certain criteria. Since you can easily add labels later, you can initially wait to determine the specific requirements you need in practice.

Typically, executing the playbook only takes a few seconds, and you can also observe the process live in the web interface while a playbook runs. With `hcloud server list`, you can find out the IP address of the new server after you successfully create it and then log in for testing with this:

```
$ ssh -i ~/.ssh/hetzner root@<IP_ADDRESS>
```

The `hetzner.hcloud` collection includes a variety of additional modules, and you should feel free to explore the online documentation at *https://docs.ansible.com/ansible/latest/collections/hetzner/hcloud*.

13.3.5 Inventorying Cloud Servers

To work effectively with provisioned cloud servers using Ansible, you'll need to add new servers to the *inventory* file first and then remove deleted servers. That is tedious, error-prone, and above all, not fun.

Thankfully, Ansible has dynamic inventory plug-ins, and the Hetzner community has, of course, provided one. The mechanism is as follows: instead of creating a static inventory file, you create a YAML file, which in the case of Hetzner must end with *hcloud.(yml|yaml)*. To test this, you place the following file in the root folder of the project:

```
---
plugin: hetzner.hcloud.hcloud
```

Listing 13.5 "inv1.hcloud.yml": Dynamic Inventory File

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

In the *ansible.cfg*, the following two changes or adjustments would be very useful:

```
# [defaults]
inventory = /dev/null
private_key_file = ~/.ssh/hetzner
# [...]
```

Listing 13.6 "ansible.cfg": Excerpt

You should avoid setting a default inventory at the start. If you later confirm that only one inventory exists, you can add it at that point.

The following command should now list all the servers of this project:

```
$ ansible-inventory -i inv1.hcloud.yml --list --yaml
```

Now, all that's needed is the proper configuration of the servers to initiate `ansible` or `ansible-playbook`. For this purpose, using `group_vars` parameters is ideal, as follows:

```
---
ansible_user: root
```

Listing 13.7 "group_vars/all.yml": Parameterization of Cloud Server

Now, you can thoroughly test whether your servers are accurately inventoried. It's crucial that the second run proceeds smoothly:

```
$ ansible -i inv1.hcloud.yml all -m ping
$ ansible -i inv1.hcloud.yml all -a "test -r /etc/shadow"
```

13.3.6 Further Possibilities of the Inventory Plug-in

You can make many more settings in the Inventory plug-in's configuration file. Here, we'll only highlight two options:

- You can provide your access token with the `api_token` directive in this scenario as well, if environment variables are not feasible:

  ```
  ---
  plugin: hetzner.hcloud.hcloud
  api_token: "{{ lookup('passwordstore', 'ansible/hcloud/api_token') }}"
  ```

  Listing 13.8 "inv2.hcloud.yml": Dynamic Inventory File with "api_token"

- You can also create dynamic groups with the `keyed_groups` directive:

  ```
  ---
  plugin: hetzner.hcloud.hcloud

  keyed_groups:
  ```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
  - key: image_os_flavor
    prefix: os
  - key: status
    prefix: server_status
  - key: labels
    prefix: label
```

Listing 13.9 "inv3.hcloud.yml": Dynamic Inventory File with Group Creation

This would then give you groups like `os_debian` and `label_type_test`, depending on your setup:

```
$ ansible-inventory -i inv3.hcloud.yml --graph
@all:
  |--@ungrouped:
  |--@hcloud:
  |  |--srv01
  |--@os_debian:
  |  |--srv01
  |--@server_status_running:
  |  |--srv01
  |--@label_type_test:
  |  |--srv01
  |--@label_purpose_learning:
  |  |--srv01
```

These are by no means all the possibilities. If you want to work more extensively with the Hetzner cloud inventory, please refer to the official plug-in documentation at *https://docs.ansible.com/ansible/latest/collections/hetzner/hcloud/hcloud_inventory.html*.

13.4 Amazon Web Services Elastic Compute Cloud

Amazon's *Elastic Compute Cloud* (EC2) is certainly one of the most popular IaaS platforms ever. If you want to follow the upcoming topics live, you need to register with Amazon Web Services (AWS) or have the appropriate credentials on hand to log in at *https://console.aws.amazon.com* and administer there.

13.4.1 Preparations on the Control Host

First, you need to perform the following preparatory steps on the control host:

1. *Please* start a new Ansible project to keep it distinct from our past efforts. For ease of use, refer to my script from Chapter 19, Section 19.1.2:

```
$ init-ansible-project.sh ~/ansible/projects/aws_ec2 && cd $_
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

2. Generate a dedicated *aws_ec2|aws_ec2.pub* SSH key pair for managing the machines in the Amazon cloud. The storage location should be outside the project in the usual *~/.ssh/* folder, and we can initially skip a passphrase in the lab:

```
$ ssh-keygen -f ~/.ssh/aws_ec2 -N ""
Generating public/private rsa key pair.
[…]
```

3. Install the *boto3* and *botocore* Python modules, which are required by the Ansible modules we want to use below. You can easily do this in our virtualenv environment:

```
$ pip3 install boto3 botocore
```

4. The most important thing is installing the `amazon.aws` collection:

```
$ ansible-galaxy collection install amazon.aws
```

5. I would also recommend that you install the AWS command line interface (CLI) tool (*https://aws.amazon.com/de/cli/*) because it allows you to quickly access information ad hoc, much faster than with a corresponding Ansible call:

```
$ pip3 install awscli
```

13.4.2 Preparations in the Cloud

I now assume that you're logged in to the Amazon cloud console. You'll need to perform the following steps.

1. Create an IAM user to manage resources in the Amazon cloud via API. In the top search field, search for `IAM` and select the appropriate result to go to the IAM dashboard.
2. On the left, select the **Users** menu item, click on **Create User**, and assign a name—`ansible` would certainly be a good choice.
3. On the next page, select **Attach policies directly**. Then, assign the `AmazonEC2FullAccess` permission, which you can best locate by using the search field. Check the box and then click on **Create User**.
4. On the list of users that appears, click on the new user. This will take you to the user's information or overview page.
5. Click the **Create Access Key** link at the top.

   Then, you select the **Application running outside of AWS** use case and then **Next and create access key**. By clicking on **Show**, you now have the option to copy and save the access key ID and the secret access key.

   Please do this *immediately* because you will never be able to view the secret key again. You need to put both pieces of information into the password store, like this:

```
$ pass insert ansible/aws_ec2/access_key
$ pass insert ansible/aws_ec2/secret_key
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The `aws` command line interface (CLI) tool can use the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables. You should test it and also specify a default region:

```
$ export AWS_ACCESS_KEY_ID=$(pass ansible/aws_ec2/access_key)
$ export AWS_SECRET_ACCESS_KEY=$(pass ansible/aws_ec2/secret_key)
$ export AWS_DEFAULT_REGION=eu-central-1

$ aws ec2 describe-instances
{
    "Reservations": []
}
```

13.4.3 Using Cloud Modules

For example, we'll begin by simply querying the available regions. The following playbook achieves this:

```
---
- hosts: localhost

  tasks:
    - name: Gather information about all regions
      amazon.aws.aws_region_info:
      register: info

    - debug: var=info
```

Listing 13.10 "info.yml": Gathering Information about Regions

However, this variant is only valid if all three relevant environment variables are set correctly, as follows:

```
$ export AWS_ACCESS_KEY_ID=$(pass ansible/aws_ec2/access_key)
$ export AWS_SECRET_ACCESS_KEY=$(pass ansible/aws_ec2/secret_key)
$ export AWS_DEFAULT_REGION=eu-central-1

$ ansible-playbook info.yml
```

The alternative is to specify the following three attributes for each corresponding task:

- `access_key`
- `secret_key`
- `region`

You can apply the same strategies that we already discussed in Section 13.3.3. I won't repeat them here.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

13.4.4 Provisioning Cloud Servers

Now, we'll assume that the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` environment variables are set correctly.

With the following playbook, you can create an initial "small" server in the Amazon cloud:

```
---
- hosts: localhost

  tasks:
    - name: Create security group
      amazon.aws.ec2_group:
        name: sg1
        description: Grant SSH access
        rules:
          - proto: tcp
            ports: [22]
            cidr_ip: 0.0.0.0/0
            rule_desc: Allow SSH
          - proto: icmp
            ports: [-1]
            cidr_ip: 0.0.0.0/0
            rule_desc: Allow ICMP

    - name: Upload SSH-key
      amazon.aws.ec2_key:
          name: sshkey_aws_ec2
          key_material: "{{ lookup('file', '~/.ssh/aws_ec2.pub') }}"

    - name: Create Ubuntu 24.04 instance
      amazon.aws.ec2_instance:
        name: srv01
        key_name: sshkey_aws_ec2
        instance_type: t2.nano
        security_group: sg1
        state: running
        network_interfaces:
          - assign_public_ip: true
        # See https://cloud-images.ubuntu.com/locator/ec2/
        image_id: ami-0731f252d4fff06e8
        tags:
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
        os: ubuntu
        type: test
        purpose: learning
```

Listing 13.11 "provisioning.yml": Provisioning Cloud Server

As you can see, it is not entirely trivial. You need to define firewall rules (otherwise, the server would not be accessible from the outside at all), and you need to provide the SSH key, which the provisioning process should then store on the new server. Note that these are one-off actions that you could also incorporate into a different playbook or perform manually beforehand.

Perhaps the most perplexing aspect of the `ec2_instance` task is the `image_id`, which isn't very informative. Amazon provides a vast array of images, and you can even add your own, so the only way to refer to them here is by using a unique ID.

Tags are technically not necessary, but in practice, they can greatly contribute to the clarity of the project later if you want to sort or group your servers according to certain criteria. Since you can easily add tags later, you can initially wait to see what specific requirements arise in practice.

The playbook run will take some time. If everything goes well, you can then find the IP address of the new server in the web interface, or you can use the *quick and dirty* method of extracting it from the JSON info clutter:

```
$ aws ec2 describe-instances | grep PublicIpAddress
                    "PublicIpAddress": "<A.B.C.D>",
```

You should now be able to log in for testing:

```
$ ssh -i ~/.ssh/aws_ec2 ubuntu@<IP_ADDRESS>
```

The `amazon.aws` collection comes with many other modules, so feel free to take a look at the online documentation at *https://docs.ansible.com/ansible/latest/collections/amazon/aws*.

13.4.5 Inventorying Cloud Servers

To effectively manage provisioned cloud servers with Ansible, you must first update the *inventory* file to include new servers and remove any that have been deleted. As before, this is tedious, error-prone, and above all, not fun.

Thankfully, Ansible has dynamic inventory plug-ins, and the AWS community has, of course, provided one. The mechanism works as follows: instead of creating a static inventory file, you create a YAML file, which in the case of Amazon must end with *aws_ec2.(yml|yaml)*. To test this, you place the following file in the root folder of the project:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
plugin: amazon.aws.aws_ec2
```

Listing 13.12 "inv1.aws_ec2.yml": Dynamic Inventory File

In the *ansible.cfg*, these two changes or adjustments would be very useful:

```
# [defaults]
inventory = /dev/null
private_key_file = ~/.ssh/aws_ec2
# [...]
```

Listing 13.13 "ansible.cfg": Excerpt

Now, the following command should already list all servers of this project. (Please be patient.):

```
$ ansible-inventory -i inv1.aws_ec2.yml --list --yaml
```

Now, all you need to do is properly configure the servers to initiate `ansible` or `ansible-playbook`. The ideal way to do this is by using `group_vars` parameters, as follows:

```
---
ansible_user: ubuntu
ansible_become: yes
```

Listing 13.14 "group_vars/all.yml": Parameterization of Cloud Server

Currently, we are assuming that only Ubuntu systems are in use because other distribution images likely have different account names. Amazon documents this at *https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/connection-prereqs-general.html*.

Now, you can thoroughly test whether your servers are correctly inventoried, as follows. (Note that it's crucial that the second run proceeds smoothly.):

```
$ ansible  -i inv1.aws_ec2.yml all -m ping
$ ansible  -i inv1.aws_ec2.yml all -a "test -r /etc/shadow"
```

13.4.6 Further Possibilities of the Inventory Plug-in

You can make many more settings in the inventory plug-in's configuration file. Here, we'll only highlight three options:

- You can provide your credentials with the `aws_access_key` and `aws_secret_key` directives if environment variables are not feasible. You can also set your regions:

  ```
  ---
  plugin: amazon.aws.aws_ec2
  ```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
aws_access_key: "{{ lookup('passwordstore', 'ansible/aws_ec2/access_key') }}"
aws_secret_key: "{{ lookup('passwordstore', 'ansible/aws_ec2/secret_key') }}"

regions:
  - eu-central-1
```

Listing 13.15 "inv2.aws_ec2.yml": Dynamic Inventory File

- You can create dynamic groups with the `keyed_groups` directive:

```
---
plugin: amazon.aws.aws_ec2

keyed_groups:
  - key: tags
    prefix: tag
```

Listing 13.16 "inv3.aws_ec2.yml": Dynamic Inventory File with Group Formation

This would then give you groups like `tag_os_ubuntu` and `tag_type_test`, depending on your setup:

```
$ ansible-inventory -i inv3.aws_ec2.yml --graph
@all:
  |--@ungrouped:
  |--@aws_ec2:
  |  |--ec2-3-70-177-176.eu-central-1.compute.amazonaws.com
  |--@tag_Name_srv01:
  |  |--ec2-3-70-177-176.eu-central-1.compute.amazonaws.com
  |--@tag_type_test:
  |  |--ec2-3-70-177-176.eu-central-1.compute.amazonaws.com
  |--@tag_os_ubuntu:
  |  |--ec2-3-70-177-176.eu-central-1.compute.amazonaws.com
  |--@tag_purpose_learning:
  |  |--ec2-3-70-177-176.eu-central-1.compute.amazonaws.com
```

- At this point, your patience may have been tested by the wait times for each call. Inventory caching can help alleviate this issue:

```
---
plugin: amazon.aws.aws_ec2

cache: yes
cache_plugin: jsonfile
cache_timeout: 3600
cache_connection: ~/.ansible/inventory_cache
```

Listing 13.17 "inv4.aws_ec2.yml": Dynamic Inventory File with Caching Enabled

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

This is very reminiscent of the facts cache you already know, and it offers similar possibilities.

What we've explored in this section isn't everything by a long shot. If you want to delve deeper into the AWS_EC2 inventory, be sure to check out the official plug-in documentation at *https://docs.ansible.com/ansible/latest/collections/amazon/aws/aws_ec2_inventory.html*.

13.5 Proxmox Virtual Environment

In this final section of the chapter, we'll take a look at the *Proxmox Virtual Environment* (Proxmox VE), which has become increasingly popular and better known in recent years. It is an open-source virtualization platform that relies on Kernel-based Virtual Machine (KVM) for virtual machines and Linux Containers (LXC) for container technology.

For this discussion, I will assume that you already have or even manage a Proxmox environment. Unlike many of our previous topics, we cannot easily simulate or virtualize this. A single server will be entirely adequate for our purposes. If you'd like to experiment and you have the necessary hardware on hand, then it will be quite straightforward for you to perform a basic installation. See *https://www.proxmox.com/en/proxmox-virtual-environment/get-started*.

13.5.1 Preparations on the Control Host

We'll also assume that you are in a virtualenv environment here (as described in Section 13.1) You perform the preparations by following these steps:

1. *Please* start a new Ansible project to keep it distinct from our past efforts. For ease of use, refer to my script from Chapter 19, Section 19.1.2:

   ```
   $ init-ansible-project.sh ~/ansible/projects/proxmox && cd $_
   ```

2. Install the *proxmoxer* Python module, which is required by the Ansible modules we want to use in the following sections:

   ```
   $ pip3 install proxmoxer
   ```

3. The actual Proxmox modules come from the `community.general` collection, which I previously recommended for installation. Just to be sure, here it is again:

   ```
   $ ansible-galaxy collection install community.general
   ```

13.5.2 Preparations in the Proxmox Virtual Environment

I now assume that you are logged into the Proxmox web interface with administrator permissions. You perform the preparations in the Proxmox VE by following these steps:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

1. Create a user for Ansible access. (You could also use the root user, but it makes sense to separate them.)

   Follow menu path **Datacenter • Permissions • Users • Add**. I suggest that you use the username `ansible`, and you don't need to change or specify anything else.

2. Click on **Permissions** one level up in the tree structure, then select **Add • User Permission** and configure the / path + `ansible@pam` user + `Administrator` role combination.

3. Add an API token (which still under **Permissions**). The user is `ansible@pam`, and the token ID could be something like `ansible`.

   Then, save the token immediately because you won't be able to view it later. We store such credentials as usual in our password store:

   ```
   $ pass insert ansible/proxmox/api_token
   ```

4. The token also needs sufficient permissions now, so you select **Permissions** again but this time **Add • API Token Permission**.

   Then, you configure the / path + your API token + `Administrator` role combination.

13.5.3 An Initial Test

Now, everything should be ready for you to test the collaboration of Ansible with Proxmox. Let's use a harmless module for information gathering:

```
---
- hosts: localhost

  vars:
    api_host: pve.example.org
    api_user: ansible@pam
    api_token_id: ansible
    api_token_secret: >-
      {{ lookup('passwordstore', 'ansible/proxmox/api_token') }}

  tasks:
    - community.general.proxmox_node_info:
        api_host:         "{{api_host}}"
        api_user:         "{{api_user}}"
        api_token_id:     "{{api_token_id}}"
        api_token_secret: "{{api_token_secret}}"
      register: node_info

    - debug: var=node_info
```

Listing 13.18 "pve_info.yml": Initial Playbook to Test Proxmox Connection

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Of course, you must define the `api_host` according to your environment. If it works, it will look something like this:

```
$ ansible-playbook pve_info.yml
[…]
ok: [localhost] =>
  node_info:
    changed: false
    failed: false
    proxmox_nodes:
    - id: node/pve
      level: ''
      node: pve
      ssl_fingerprint: 4E:24:C6:AD:5C:1B:[…]
      status: online
      type: node
[…]
```

Unfortunately, the Proxmox modules currently don't offer an alternative with environment variables. You can apply the same strategies that we already discussed in Section 13.3.3, and I won't repeat them here.

13.5.4 Provisioning of Virtual Machines

As a pure private cloud, Proxmox does not provide preprepared images, so unfortunately, you can't just quickly conjure up a small Ubuntu VM. While there are online sources of premade images, you need to ensure their trustworthiness before using them.

Typically, you'll create your own collection of "golden images" or templates, which you can then clone as needed. However, you can also personalize the approaches. I can show you a setup from my home lab; perhaps it will inspire some ideas for your own solution.

First, I have a small inventory with my own golden VM images:

```
[templates]
debian  ansible_host=192.168.1.190 vmid=100
rocky   ansible_host=192.168.1.191 vmid=101
suse    ansible_host=192.168.1.192 vmid=102
ubuntu  ansible_host=192.168.1.193 vmid=103

[templates:vars]
ansible_user=root
```

Listing 13.19 "templates": Demo Inventory with Templates

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

And with this interactive playbook, you can clone, start, and further customize an image:

```
---
- hosts: "{{template}}"
  gather_facts: false

  vars_prompt:
    - name: template
      prompt: Enter template (debian|rocky|suse|ubuntu)
      private: false
    - name: newid
      prompt: Enter new PVE ID
      private: false
    - name: newname
      prompt: Enter new PVE inventory name
      private: false
    - name: memory
      prompt: Enter memory in MB
      default: 2048
      private: false
    - name: hostname
      prompt: Enter Linux hostname
      private: false
    - name: octet
      prompt: Enter last octet of 192.168.1.xxx
      private: false
    - name: pool
      prompt: Enter pool
      private: false
      default: Testing

  module_defaults:
    community.general.proxmox_kvm:
        node:     pve
        api_host: proxmox.example.org
        api_user: ansible@pam
        api_token_id: ansible
        api_token_secret: >-
          {{ lookup('passwordstore', 'ansible/proxmox/api_token') }}

  tasks:
    - name: Clone {{template}} template
      delegate_to: localhost
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    community.general.proxmox_kvm:
      clone: clone # If vmid is set, clone can take an arbitrary value
                   # but is required for initiating the clone.
      vmid:  "{{ hostvars[template]['vmid'] }}"
      newid: "{{newid}}"
      name:  "{{newname}}" # Proxmox-name
      pool:  "{{pool}}"
      timeout: 600
      state: present
    register: clone

  - pause:
      seconds: 10

  - name: Change some settings
    delegate_to: localhost
    community.general.proxmox_kvm:
      vmid:   "{{newid}}"
      memory: "{{memory}}"
      update: true


  - name: Start VM
    delegate_to: localhost
    community.general.proxmox_kvm:
      vmid: "{{newid}}"
      state: started

  - wait_for_connection:

  - name: Set hostname
    hostname:
      name: "{{hostname}}"

  - name: Set IPv4 address
    nmcli:
      conn_name: "wired-1"
      ip4: "192.168.1.{{octet}}"
      state: present

  - name: Reboot and end play
    shell: "sleep 1 && shutdown -r now"
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    async: 1
    poll: 0
    failed_when: false
```

Listing 13.20 "provision.yml": Playbook for Cloning VM

You can then request a new clone is with this:

```
$ ansible-playbook -i ../templates provision.yml
```

The customization process after cloning is highly individualized. For instance, all my templates already include my SSH key, allowing for seamless access. Furthermore, a NetworkManager is running, and it enables me to standardize the IP configuration. Finally, as I mentioned, you should use this example merely as a source of inspiration!

13.5.5 Outlook

There are currently sixteen Proxmox modules in the `community.general` collection, and of course, there's also an inventory plug-in that we haven't discussed yet. However, I hope that through our experiments, we have laid enough groundwork so that you can continue on your own. I wish you great success with your Ansible journey in the cloud—whether it's public or private!

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 14
Ansible as an Orchestration Tool

Ansible can be defined as an open-source automation tool for orchestration and general configuration and administration of computers. Since I'm mentioning the term *orchestration* here for the first time in this book, it may seem to you like I've kept something important from you until now! Since this term might not be a familiar one even for some experienced IT professionals, let's first clarify its meaning. Wikipedia explains it thusly (see *https://en.wikipedia.org/wiki/Orchestration_(computing)*):

> *Orchestration (in computing) ... is about aligning the business request with the applications, data, and infrastructure.*

But this definition still doesn't make orchestration entirely clear. If you use Ansible to install a web application with a frontend and a database, have you already orchestrated it? And is it aligned?

Red Hat has also dedicated an article to the topic. The introduction (at *https://www.redhat.com/en/topics/automation/what-is-orchestration*) reads as follows:

> *Orchestration is the coordinated execution of multiple IT automation tasks or processes. Orchestration is usually applied across multiple computer systems, applications, and services to ensure that deployment, configuration management, and other processes are performed in the proper sequence.*

You'll find similar definitions in many other places, and with regard to Ansible, you could quite rightly say that we're already doing all of that! However, when it comes to *coordination*, we admittedly lack a couple of techniques to ensure that everything is done in the right order across multiple participating systems. Think, for example, of a set of systems behind a load balancer that need to update its software. The desired automation could gradually remove some machines from the load balancing and monitoring, perform the update, reboot if necessary, and finally add the systems back into load balancing and monitoring—until all systems are updated.

I would like to introduce you to several techniques that can help you manage these requirements effectively.

14.1 Many Target Hosts for Testing

Some of the techniques I present here only matter when you are dealing with many target hosts. Now of course, "many" is relative, but it's certainly more than four. With VMs,

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

you quickly reach the natural limit of our lab; starting up ten Vagrant VMs at once is a tough requirement. Of course, there's still the cloud, but cloud servers cost money, and you might also encounter surprising limits. For example, at Hetzner, there is a server limit for new customers that is significantly less than ten. If you want to exceed it, you need to contact support. (This isn't a big deal, but you still need to do it.)

The good news is that for what we want to test in this chapter, a single server is enough and we can simply "duplicate" it.

You should start by creating a new Ansible project for our endeavor. For simplicity's sake, you should use my script from Chapter 19, Section 19.1.2:

```
$ init-ansible-project.sh ~/ansible/projects/orchestration && cd $_
```

With the following inventory, you can now simulate ten target systems, which are actually all `localhost`:

```
[test]
dummy[01:10] ansible_connection=local
```

Listing 14.1 "inventory": Inventory with Ten Simulated Target Hosts

With that, the whole group should be ready:

```
$ ansible-inventory --graph
@all:
  |--@ungrouped:
  |--@test:
  |  |--dummy01
  |  |--dummy02
  |  |--dummy03
  |  |--dummy04
  |  |--dummy05
  |  |--dummy06
  |  |--dummy07
  |  |--dummy08
  |  |--dummy09
  |  |--dummy10

$ ansible all -m ping
[…]
```

14.2 Altering the Sequence of Execution

Let's take the following playbook as a starting point. It waits for a certain amount of time on all hosts:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- hosts: test

  vars:
    same:   yes
    factor: 1.0

  tasks:
    - set_fact:
        duration: >-
          {%- if same|bool -%}
            {{ factor|float }}
          {%- else -%}
            {{ (inventory_hostname|regex_search('\d+')|int) * factor|float }}
          {%- endif -%}

    - debug:
        msg: >-
          [INFO] {{inventory_hostname}}: Pausing for {{duration}} seconds

    - name: sleep {{duration}} seconds
      command: sleep {{duration}}
      changed_when: false

    - debug:
        msg: >-
          [INFO] {{inventory_hostname}}:
          Done at {{lookup('pipe', 'date +%T')}}
```

Listing 14.2 "tasks.yml": Playbook with Simulated Task Durations for Experimenting

By default (`same=yes`), all hosts wait for 1 second. With `same=no`, the duration depends on the host number: host `py01` would wait for 1 second, host `py02` would wait for 2 seconds, and so on. You can use the `factor` to extend or reduce these durations as needed.

We are familiar with Ansible's default behavior when executing playbooks: for each task, it waits for all hosts to complete, and it "tasks" five hosts in parallel. (We discussed this a while ago in Chapter 3, Section 3.4.1.)

With a standard fork level of `5`, you will observe the following behavior in the middle section after starting the playbook:

```
$ ansible-playbook tasks.yml
[…]
TASK [sleep 1.0] ***********************************************************
ok: [dummy02]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
ok: [dummy05]
ok: [dummy01]
ok: [dummy03]
ok: [dummy04]
... short pause ...
ok: [dummy06]
ok: [dummy07]
ok: [dummy10]
ok: [dummy08]
ok: [dummy09]
[…]
```

Increasing the fork level offers a performance advantage, which becomes more noticeable the longer the hosts take:

```
$ time ansible-playbook tasks.yml
real    0m5.869s
$ time ansible-playbook tasks.yml -f 10
real    0m4.570s
$ time ansible-playbook tasks.yml -e factor=2
real    0m7.704s
$ time ansible-playbook tasks.yml -e factor=2 -f 10
real    0m5.662s
```

But as I mentioned, this is nothing new. So, let's explore how we can get Ansible to adopt entirely different processing strategies..

14.2.1 "throttle" and "order"

There are various reasons why you might want to arrange the hosts in a specific order during processing. You must first be fundamentally aware that *order* and *parallelism* do not go well together. *Order* generally means that certain hosts wait until other hosts are finished—in other words, it means there is sequential processing.

If you need a host order in a specific task, you can work with the `throttle` directive to limit the number of parallel workers. The `throttle = 1` directive means fully sequential, and with the global `order` directive, you can also define the type of start order if you need to. The really practical values here are `sorted`, `reverse_sorted`, and `shuffle`.

If you want to try it, you can make a copy of the playbook from Listing 14.2 and change it in two places:

```
---
- hosts: test
  order: sorted    # <===== !
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
[…]

    - command: sleep {{duration}}
      changed_when: false
      throttle: 1  # <===== !

[…]
```

Listing 14.3 "tasks-throttle.yml": Example Task with "throttle" Directive

```
$ time ansible-playbook tasks-throttle.yml
TASK [sleep 1.0] ************************************************************
All nicely one after the other in a second interval:
ok: [dummy01]
ok: [dummy02]
ok: [dummy03]
ok: [dummy04]
ok: [dummy05]
ok: [dummy06]
ok: [dummy07]
ok: [dummy08]
ok: [dummy09]
ok: [dummy10]
```

14.2.2 "serial"

The `serial` directive is applied at the play level. In the simplest case, the value is a simple number N, and for Ansible, this means "Take N hosts at a time and complete the entire play with them, then take the next N hosts, and so on." The fork level is still respected. Such a group of hosts in serial mode is also often referred to as a *batch* in Ansible jargon.

With a `serial` value of `1`, you would have a sequential execution during the entire play. An excerpt of the original *tasks.yml* playbook would look like this:

```
---
- hosts: test
  serial: 1
  […]
```

Listing 14.4 "tasks-serial.yml": Application of "serial"

When executing, it looks something like this:

```
$ ansible-playbook tasks-serial.yml | grep INFO
  msg: '[INFO] dummy01: Pausing for 1.0 seconds'
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
  msg: '[INFO] dummy01: Done at 13:16:56'
  msg: '[INFO] dummy02: Pausing for 1.0 seconds'
  msg: '[INFO] dummy02: Done at 13:16:58'
  msg: '[INFO] dummy03: Pausing for 1.0 seconds'
  msg: '[INFO] dummy03: Done at 13:16:59'
[…]
```

You could certainly influence the order again with `order`, and if you didn't, the run would be reproducible but usually not predictable. (The order is determined by how Ansible processes its inventory sources internally.)

When looking toward larger environments and rolling updates, it's useful that you can specify a percentage instead of a number. With the following setting, Ansible would process 30% of all hosts at a time:

```
---
- hosts: test
  serial: "30%"
  […]
```

Listing 14.5 "tasks-serial.yml": Application of "serial" with Percentage

You can even specify lists of values with `serial`. With the following setting, Ansible would first process one host, then two, and finally batches of five hosts:

```
---
- hosts: test
  serial:
    - 1
    - 2
    - 5
  […]
```

Listing 14.6 "tasks-serial.yml": Application of "serial" with List of Values

Of course, this also works with percentages—even those that are mixed with simple numbers.

14.2.3 Faulty Hosts in "serial" Mode

Warning, Bug!

Due to a very subtle bug, which was still present at least in Ansible Core version 2.17, the `max_fail_percentage` directive that we'll discuss shortly does not work together with `force_handlers = true`.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

If you have this (generally very recommended) setting in your *ansible.cfg*, you'll have to do without it now:

```
[…]
# Does not work in combination with max_fail_percentage:
#force_handlers = true
[…]
```

Listing 14.7 "ansible.cfg"

With that clarified, let's move on to how Ansible manages errors in serial or batch mod—because of course, not everything always goes smoothly during things like a rolling update with dozens of involved servers. However, as long as not all hosts in a batch fail, the play will continue by default. In extreme cases, having a single "surviving" host per batch will be enough for the play to run to completion.

The following example demonstrates this: ten hosts are processed in two batches of five, and in each batch, four hosts fail and only `dummy01` and `dummy06` are successful:

```
---
- hosts: test
  serial: 5
  order: sorted

  tasks:
    - fail:
      when: inventory_hostname not in ['dummy01', 'dummy06']

    - debug: msg="{{inventory_hostname}} finished successfully."
```

Listing 14.8 "tasks-serial-fail1.yml": Simulated Failure of Hosts

However, you can limit the number of allowed failed hosts with the `max_fail_percentage` directive. In the above example, four out of five hosts fail, which is 80%. For instance, you could set this value to 30%:

```
---
- hosts: test
  serial: 5
  order: sorted
  max_fail_percentage: 30
  […]
```

Listing 14.9 "tasks-serial-fail2.yml": Application of "max_fail_percentage"

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Indeed, it ends after the first batch:

```
$ ansible-playbook tasks-serial-fail2.yml
TASK [fail] ********************************************************************
skipping: [dummy01]
fatal: [dummy02]: FAILED! => changed=false
  msg: Failed as requested from task
fatal: [dummy03]: FAILED! => changed=false
  msg: Failed as requested from task
fatal: [dummy04]: FAILED! => changed=false
  msg: Failed as requested from task
fatal: [dummy05]: FAILED! => changed=false
  msg: Failed as requested from task

NO MORE HOSTS LEFT *************************************************************
```

It's also noteworthy that the value of `max_fail_percentage` *must be exceeded* for the play to end. So, you will get an error if you specify 79 in the example, but you won't if you specify 80.

14.2.4 Strategy Plug-ins

We haven't been able to change one behavior pattern of Ansible so far: all involved hosts must complete a task before moving on to the next task. (This is even true for a `serial` value of 1—then, "all hosts" are just one each.) This can become quite apparent when there's a significant disparity in the processing speeds of the hosts: slower hosts end up holding back faster ones.

However, you can also configure this behavior in Ansible. By choosing a so-called *strategy plug-in*, you can choose from several behaviors. Essentially, you have three strategy plug-ins available:

1. The `linear` strategy plug-in is the default.
2. With the `free` strategy plug-in, a host can immediately proceed to the next task when it finishes the current one.
3. The `host_pinned` strategy plug-in is similar to `free` (see the following sections).

You can set the strategy in the play with the `strategy` directive or with the `ANSIBLE_STRATEGY` environment variable. You can find information about this in the online documentation at *https://docs.ansible.com/ansible/latest/plugins/strategy.html*.

The "linear" Strategy

Let's start with the well-known standard execution strategy, in which all hosts must complete a task before the next task can begin:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
$ export ANSIBLE_STRATEGY=linear
$ ansible-playbook tasks.yml -e same=no -f 5 | grep INFO
  msg: '[INFO] dummy01: Pausing for 1.0 seconds'
  msg: '[INFO] dummy02: Pausing for 2.0 seconds'
  msg: '[INFO] dummy03: Pausing for 3.0 seconds'
  msg: '[INFO] dummy04: Pausing for 4.0 seconds'
  msg: '[INFO] dummy05: Pausing for 5.0 seconds'
  msg: '[INFO] dummy06: Pausing for 6.0 seconds'
  msg: '[INFO] dummy07: Pausing for 7.0 seconds'
  msg: '[INFO] dummy08: Pausing for 8.0 seconds'
  msg: '[INFO] dummy09: Pausing for 9.0 seconds'
  msg: '[INFO] dummy10: Pausing for 10.0 seconds'
... longer waiting time ...
  msg: '[INFO] dummy01: Done at 13:39:16'
  msg: '[INFO] dummy02: Done at 13:39:16'
  msg: '[INFO] dummy03: Done at 13:39:16'
  msg: '[INFO] dummy04: Done at 13:39:16'
  msg: '[INFO] dummy06: Done at 13:39:16'
  msg: '[INFO] dummy05: Done at 13:39:16'
  msg: '[INFO] dummy07: Done at 13:39:16'
  msg: '[INFO] dummy08: Done at 13:39:16'
  msg: '[INFO] dummy09: Done at 13:39:16'
  msg: '[INFO] dummy10: Done at 13:39:16'
```

So far, there have been no surprises: the final task is only completed once all hosts have finished, and some hosts have experienced long wait times. Increasing the fork level slightly reduces the total execution time.

The "free" Strategy

With the `free` strategy, a host can immediately proceed to the next task upon completion and does not have to wait for its "colleagues." An increased fork level has an especially positive effect:

```
$ export ANSIBLE_STRATEGY=free
$ ansible-playbook tasks.yml -e same=no -f 10 | grep INFO
  msg: '[INFO] dummy01: Pausing for 1.0 seconds'
  msg: '[INFO] dummy02: Pausing for 2.0 seconds'
  msg: '[INFO] dummy03: Pausing for 3.0 seconds'
  msg: '[INFO] dummy04: Pausing for 4.0 seconds'
  msg: '[INFO] dummy05: Pausing for 5.0 seconds'
  msg: '[INFO] dummy06: Pausing for 6.0 seconds'
  msg: '[INFO] dummy07: Pausing for 7.0 seconds'
  msg: '[INFO] dummy08: Pausing for 8.0 seconds'
  msg: '[INFO] dummy09: Pausing for 9.0 seconds'
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
  msg: '[INFO] dummy10: Pausing for 10.0 seconds'
  msg: '[INFO] dummy01: Done at 13:40:27'
  msg: '[INFO] dummy02: Done at 13:40:28'
  msg: '[INFO] dummy03: Done at 13:40:29'
  msg: '[INFO] dummy04: Done at 13:40:30'
  msg: '[INFO] dummy05: Done at 13:40:31'
  msg: '[INFO] dummy06: Done at 13:40:32'
  msg: '[INFO] dummy07: Done at 13:40:33'
  msg: '[INFO] dummy08: Done at 13:40:34'
  msg: '[INFO] dummy09: Done at 13:40:35'
  msg: '[INFO] dummy10: Done at 13:40:36'
```

In terms of the total runtime of the playbook, you won't notice any significant changes because the slowest host still drags down the overall runtime level. However, you can clearly see how individual hosts no longer have to wait and finish earlier compared to the `linear` strategy.

The "host_pinned" Strategy

The `host_pinned` strategy was only added in Ansible 2.7, and its mode of operation is not immediately apparent. With a sufficiently high fork level, it is identical to the `free` strategy—but if the fork level is lower than the number of hosts, then Ansible ensures that the number of hosts with an active play never exceeds the fork level. As soon as a host is finished, a new host that has been waiting will start immediately:

```
$ export ANSIBLE_STRATEGY=host_pinned
$ ansible-playbook tasks.yml -e same=no -f 5 | grep INFO
  msg: '[INFO] dummy01: Pausing for 1.0 seconds'
  msg: '[INFO] dummy02: Pausing for 2.0 seconds'
  msg: '[INFO] dummy03: Pausing for 3.0 seconds'
  msg: '[INFO] dummy04: Pausing for 4.0 seconds'
  msg: '[INFO] dummy05: Pausing for 5.0 seconds'
  msg: '[INFO] dummy01: Done at 13:42:05'
  msg: '[INFO] dummy06: Pausing for 6.0 seconds'
  msg: '[INFO] dummy02: Done at 13:42:06'
  msg: '[INFO] dummy07: Pausing for 7.0 seconds'
  msg: '[INFO] dummy03: Done at 13:42:07'
  msg: '[INFO] dummy08: Pausing for 8.0 seconds'
  msg: '[INFO] dummy04: Done at 13:42:08'
  msg: '[INFO] dummy09: Pausing for 9.0 seconds'
  msg: '[INFO] dummy05: Done at 13:42:09'
  msg: '[INFO] dummy10: Pausing for 10.0 seconds'
  msg: '[INFO] dummy06: Done at 13:42:11'
  msg: '[INFO] dummy07: Done at 13:42:13'
  msg: '[INFO] dummy08: Done at 13:42:15'
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
msg: '[INFO] dummy09: Done at 13:42:17'
msg: '[INFO] dummy10: Done at 13:42:19'
```

At a low fork level, what you'll see will look like quite a mess because whenever a host finishes its tasks, a new host is allowed to start immediately.

Important

After these experiments, you should reset the execution strategy to the default value:

```
$ unset ANSIBLE_STRATEGY
```

14.3 Delegation

A common pattern in orchestration is that a task belongs to one host but needs to be executed on another host. Think, for example, of hosts that need to be added to a load balancer configuration or registered in server monitoring.

The `delegate_to` directive is frequently used here, as you might remember from Chapter 6, Section 6.13 (where we only discussed delegating to the control host). In reality, you can delegate a task to any host in the inventory, even those that are not participating in the current play.

See the following playbook for a "harmless" example, in which the names and IP addresses of the `dummy01` to `dummy09` hosts are written into a text file on `dummy10`:

```
---
- hosts: test:!dummy10
  tasks:
    - name: Add the name of each host to a file on dummy10
      lineinfile:
        path: /tmp/dummyhosts.txt
        create: yes
        line: "{{inventory_hostname}} has {{ansible_default_ipv4.address}}"
        regexp: "^{{inventory_hostname}}.*"
      delegate_to: dummy10
```

Listing 14.10 "delegate_to.yml": Example for Delegation

Due to Ansible's parallel execution, there could be potential issues because multiple processes might attempt to write to the same file on the same machine at the same time. But the `lineinfile` module implemented a file-locking mechanism years ago. (Anxious individuals could, of course, still work with `serial: 1`, but the performance would be abysmal.) So, let's start it and then check the result:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
$ ansible-playbook delegate_to.yml
[…]
TASK [Add the name of each host to a file on dummy10] ***********************
changed: [dummy05 -> dummy10]
changed: [dummy02 -> dummy10]
changed: [dummy01 -> dummy10]
changed: [dummy03 -> dummy10]
changed: [dummy04 -> dummy10]
changed: [dummy06 -> dummy10]
changed: [dummy07 -> dummy10]
changed: [dummy09 -> dummy10]
changed: [dummy08 -> dummy10]
[…]

$ ansible dummy10 -a "cat /tmp/dummyhosts.txt"
dummy10 | CHANGED | rc=0 >>
dummy05 has 10.0.2.15
dummy02 has 10.0.2.15
dummy01 has 10.0.2.15
dummy03 has 10.0.2.15
dummy04 has 10.0.2.15
dummy06 has 10.0.2.15
dummy07 has 10.0.2.15
dummy09 has 10.0.2.15
dummy08 has 10.0.2.15
```

Of course, the addresses here are all the same—it's always the same host due to our inventory trick. However, if you're wondering now whether there's a more direct way to do this, the answer in this case is yes. You could generate the file directly with a bit of Jinja magic. You'd create the content with a loop over the nine hosts and access the respective host facts:

```
---
# Caution: This playbook is not correct under all circumstances!

- hosts: dummy10
  tasks:
    - name: Add the name of each host to a file
      copy:
        dest: /tmp/dummyhosts.txt
        content: |
          {% for h in groups['test'] if not h == 'dummy10' %}
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
        {{ hostvars[h]['inventory_hostname'] ~ ' has ' ~
           hostvars[h]['ansible_default_ipv4']['address'] }}
        {% endfor %}
```

Listing 14.11 "dummylist-v1.yml": First, Not Entirely Correct Idea

This looks really clever . . . but only as long as a valid facts cache exists. How else should `dummy10` know facts about the other nine hosts? Try deleting the facts cache and the playbook will fail:

```
$ rm -f ~/.ansible/fact_cache/dummy*

$ ansible-playbook dummylist-v1.yml
TASK [Add the name of each host to a file] ************************
  msg: |-
    The task includes an option with an undefined variable. The error was:
    'ansible.vars.hostvars.HostVarsVars object' has no attribute
    'ansible_default_ipv4'
    […]
```

You can only fix this now by first gathering the necessary facts, because you generally cannot rely on the existence of a facts cache. So, you should do something like this:

```
---
- hosts: test:!dummy10
  gather_facts: yes
  tasks: []

- hosts: dummy10
  tasks:
    - name: Add the name of each host to a file
      copy:
        dest: /tmp/dummyhosts.txt
        content: |
          {% for h in groups['test'] if not h == 'dummy10' %}
          {{ hostvars[h]['inventory_hostname'] ~ ' has ' ~
             hostvars[h]['ansible_default_ipv4']['address'] }}
          {% endfor %}
```

Listing 14.12 "dummylist-v2.yml": Correct Version with "gather_facts"

The first play with the empty task list is kind of the trick to gather the facts. For clarity, it is also mentioned again, although it would be the default.

However, if you don't want to (or can't) use two plays, it becomes quite tricky, but it's still doable:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts: dummy10

  tasks:
    - setup:
      delegate_to: "{{item}}"
      delegate_facts: yes
      with_items: "{{groups['test']}}"

    - name: Add the name of each host to a file
      copy:
        dest: /tmp/dummyhosts.txt
        content: |
          {% for h in groups['test'] if not h == 'dummy10' %}
          {{ hostvars[h]['inventory_hostname'] ~ ' has ' ~
             hostvars[h]['ansible_default_ipv4']['address'] }}
          {% endfor %}
```

Listing 14.13 "dummylist-v3.yml": Correct Version with "setup/delegate_facts"

In the first task, the necessary facts are now being gathered. But to ensure that the facts remain with the respective delegated host, you need to use the `delegate_facts` directive; otherwise, this version won't work any better than the first version.

My conclusion is that when you want to generate a file whose content depends on many different hosts, you should do it with Jinja as often as possible because it is by far the most efficient method. However, make sure that the necessary facts are available!

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 15
Ansible and Windows

In this chapter, we take a trip into the Microsoft world. Since version 2.1, Ansible has had the capability to manage Windows targets, which is certainly a very welcome development.

The control host will primarily be a Linux system, but we will briefly explore the options available if you prefer to run Ansible directly on a Windows system. If you're not interested in this, you can, of course, skip this part and go straight to Section 15.2.

15.1 A Control Host Based on Windows

We can have this if we must . . . but why, really? Simply having a Linux VM with VirtualBox, VMware, or HyperV is completely sufficient, isn't it?

That's certainly for die-hard Linux users, but there are other people out there. The day may come when you'll want to give a playbook to a Windows user. ("Just let it run..."). But how? Do you want to give them another three days of introduction to Linux and virtualization right away? And how can they then adjust their Playbook painlessly and without using tricks? With `vi`?

Alright, let's just assume that there is a need. I should note up front that Ansible can only operate in a "Linux-like" environment, so you won't find a simple installer in a familiar form like "*ansible-setup-x86_64.exe*" anywhere. Essentially, you have two more-or-less viable options to run Ansible directly on Windows:

- Windows Subsystem for Linux (WSL)
- Cygwin

If you have the choice and you've met all technical requirements, I recommend that you use WSL. Version 2 offers good integration and impressive performance.

15.1.1 The Windows Subsystem for Linux

This is a compatibility layer for running Linux programs, and it's been available since Windows 10. Specifically, we're looking at WSL Version 2, which brings significant performance advantages. You'll probably find that compatibility issues with other virtualization solutions like VirtualBox and VMware have been resolved by now if you're running the latest versions of all components involved.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

To use the installation method described here, you need at least Windows 10 Version 2004. You should enter `winver` in a command prompt if you're unsure about your version.

Installation

You can perform the installation by following these two simple steps (from the manufacturer's page at *https://docs.microsoft.com/en-us/windows/wsl/install*):

1. Open a command prompt in administrator mode (right-click **Run as administrator**) and enter the following command:

   ```
   wsl --install
   ```

2. Restart your computer to complete the WSL installation.

The WSL installer also sets up Ubuntu 24.04. When you start it for the first time, you need to create a user, which will then have unrestricted `sudo` rights. If you're not a fan of Ubuntu, you can choose from many other distributions; in a Windows command prompt, you can see the entire list with this:

```
wsl --list --online
```

Now, you can install a Debian as an example with this:

```
wsl --install -d Debian
```

From a command prompt, you can now start your Linux environment at any time with commands like `ubuntu` or `debian`. You can see all installed distributions with this:

```
wsl --list --all
```

Cross-Platform Work

- **File access Linux → Windows**
  The Windows drives are mounted under */mnt* (e.g., */mnt/c/*).
- **File access Windows → Linux**
  You can enter `\\wsl$` in Explorer to access the active Linux systems. In a Linux system, `explorer.exe .` is also a handy command.

Almost Forgot: Ansible

Before we completely forget why we did all this, you still need to install Ansible in WSL-Linux. For example, you can do it on a Debian-like system with this:

```
$ sudo apt install ansible
```

Using PuTTY Keys in Windows Subsystem for Linux

You can access the PuTTY agent from WSL, but I find the solutions too complicated. If you want to use a PuTTY key in the WSL context, I recommend a simple key export via

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

PuTTYgen. You can create the *.ssh/* directory in the Linux home folder, and inside it, you can do the following:

- Place a new *id_rsa.pub* file with the public key that you see in PuTTYgen above after loading your key.
- Place a new *id_rsa* file via the **Conversions • Export OpenSSH key** PuTTYgen menu item.

Then, you just need to adjust the permissions of the private key file (e.g., `chmod 600 id_rsa`). After that, you can start an SSH agent with a procedure that will be familiar to you from the Linux world and feed it your key:

```
$ eval $(ssh-agent)
$ ssh-add
```

15.1.2 Cygwin

Cygwin is a huge collection of open-source tools, all of which have been natively translated for Windows to give the overall feel of a typical Linux environment. However, you can't simply run a native Linux application under Windows, even with Cygwin (which is not a problem in our use case at the moment).

Cygwin has been around forever, and it has no special installation requirements.

Installation

You can download the appropriate installer from *https://cygwin.com* (which is usually at *https://cygwin.com/setup-x86_64.exe*).

The installer also includes the package manager, so you'll need it from time to time. If you've misplaced it, you can of course download and start it again at any time.

After you answer basic questions and select a mirror server, the package selection window will appear. If you confirm twice with **Next**, you'll receive a basic installation, which may take a while.

After you complete the installation, you can start a Cygwin terminal. This gives you a Bourne Again Shell (Bash shell) with a Linux-like environment.

Cross-Platform Work

- **File access Cygwin → Windows**
  The Windows drives are mounted under */cygdrive* (e.g., in */cygdrive/c/*).
- **File access Windows → Cygwin**
  In Explorer, you can select the installation folder (if you haven't modified it, it's *C:\cygwin64*). Within the Cygwin environment, `explorer.exe .` is also a handy command.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Installing Ansible and Other Useful Software

Next, you should restart the installer. All previous settings have been saved, so you just need to confirm everything if it still fits.

The package selection window takes a bit of getting used to. First, you select the **Not Installed** setting at the top left under **View**. For each package that you search for and find, you double-click on **Skip** to display the available version. You should then search for and select the following packages as examples and then finally install them:

- *ssh-pageant* (if you want to use the PuTTY agent)
- *git*

Unfortunately, at the time of my tests, the Cygwin-Ansible package was only available in version 2.8.x, which is really not quite up-to-date. If nothing has changed in the meantime, you should install the latest community package possible with the given Python version by using `pip3`:

```
$ pip3 install ansible
[…]

$ ansible --version
ansible [core 2.11.12]
(Well, at least ...)
```

Using PuTTY Keys in the Cygwin Environment

You can relatively easily use an *active* PuTTY agent in the Cygwin environment. You do this by simply entering the following command:

```
$ eval $(ssh-pageant)
```

After that, `ssh-add -l` should display your key(s).

SSH Issues with Git Servers

When you're connecting via SSH, you may experience issues with Git servers like Gitea and GitHub. A possible workaround is the following SSH client configuration in the home directory of the Cygwin environment:

```
Host *
    HostKeyAlgorithms +ssh-rsa
    PubkeyAcceptedKeyTypes +ssh-rsa
```

Listing 15.1 "~/.ssh/config": SSH Client Configuration

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

15.2 Windows Targets and Windows Remote Management

From this point forward, we'll focus on managing Windows systems as target hosts. From a technical perspective, a lot changes once Windows targets come into play: for one thing, *Windows Remote Management* (WinRM) is used as the transport protocol instead of SSH, and the Ansible modules for Windows are programmed in PowerShell rather than Python. The initial setup of WinRM can be challenging, but once it's configured, you can proceed as usual. The only outwardly noticeable difference will be that all Windows modules will have the `win_` prefix in their names. Thus, `ping` becomes `win_ping`, `user` becomes `win_user`, etc. This is one of the reasons why it's generally advisable to keep separate inventories for Linux and Windows environments.

The WinRM protocol isn't something even experienced Windows admins deal with daily, so here are a few brief explanations.

Windows Remote Management describes a Simple Object Access Protocol (SOAP)–based network protocol for exchanging management information. A small web server or web service, which is referred to as the WinRM HTTP listener, operates on the target system to be addressed. This essential software component has been included in all Windows versions since Vista, but it usually needs to be activated first.

WinRM uses ports 5985 (HTTP) and 5986 (HTTPS) by default. Which authentication options can be used depends on the target system: basic authentication always functions in principle, but it's limited to accessing a local admin account only. (Additionally, the transport should always be via HTTPS, because otherwise, the login information will be sent in plaintext over the network.) If the target system is a domain member, then Kerberos is also available; it's a bit more complicated to set up on the Linux side.

There are also three other methods: certificate-based, NT LAN Manager (NTLM), and Credential Security Support Provider (CredSSP) authentication, which we will not discuss here. You can find more detailed information on WRM at *https://docs.ansible.com/ansible/latest/user_guide/windows_winrm.html*.

15.3 Preparations on the Control Host

To manage Windows hosts, you need to have the Python-WinRM module on the control host. On our Debian system, you can do this is quickly or you may not even need to do it anymore:

```
$ sudo apt install python3-winrm
```

If your Linux does not provide a package for this, you must install the module via PIP. The package is called *pywinrm* there.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Next, please create a new Ansible project because you definitely don't want to mix it with the existing experiments. For simplicity's sake, you can use my script from Chapter 19, Section 19.1.2:

```
$ init-ansible-project.sh ~/ansible/projects/windows && cd $_
```

In Listing 15.2, you'll see an inventory file with an example target system that doesn't belong to a domain and that we'll therefore address with basic authentication. Instead of keeping IP addresses as parameters, you could of course also make */etc/hosts* entries again.

```
[windows]
win10.example.org  ansible_host=192.168.150.60

[windows:vars]
ansible_user=Administrator
ansible_password=PasswOrd
ansible_connection=winrm
ansible_winrm_transport=basic
ansible_winrm_server_cert_validation=ignore
```

Listing 15.2 "inventory": Example Windows Inventory

The `ansible_user` must exist on the Windows system, and it must have admin rights. You might want to create a dedicated account for it, and we'll talk more about that on the Windows side in a bit.

Since we'll be using a self-signed certificate on the target system, the setting in the last line ensures convenient (though not entirely ideal) handling of this situation.

15.4 Prerequisites on the Windows Side and Windows Remote Management Setup

Basically, with Ansible, you can manage all operating system variants that are officially supported by Microsoft. The technical requirements are straightforward: you need PowerShell version 3.0 or later and .NET Framework version 4.0 or later on the Windows targets.

If you've met these requirements, you can simplify the WinRM setup process with a PowerShell script that's provided by the Ansible project. It generates a self-signed certificate, configures WinRM accordingly, and opens the appropriate incoming port in the Windows firewall if necessary. (You can do all this manually, but it's not fun.)

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The script is called *ConfigureRemotingForAnsible.ps1*, and it's at *https://github.com/ansible/ansible-documentation/blob/devel/examples/scripts/ConfigureRemotingForAnsible.ps1*. If you want to save it from your browser, you should click on the **Raw** button before using something like **Save page as...**.

Now, you should open a Windows shell (`cmd` or `powershell`) *with administrator privileges*, navigate to the location of the download, and start the script as follows:

```
powershell -ExecutionPolicy ByPass -File ConfigureRemotingForAnsible.ps1
Self-signed SSL certificate generated; thumbprint: CF919387B8F3[...]

wxf                   : http://schemas.xmlsoap.org/ws/2004/09/transfer
a                     : http://schemas.xmlsoap.org/ws/2004/08/addressing
w                     : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
lang                  : de-DE
Address               : http://schemas.xmlsoap.org/ws/2004/08/[...]
ReferenceParameters : ReferenceParameters

OK.
```

Now, the only thing left is the user account that Ansible should use (as specified in the inventory) if it needs to. You should set this up accordingly and ensure that it has sufficient rights, either by being a member of the administrators' group or the remote management users' group.

An ad hoc test on our control host should succeed now (please note the use of `win_ping` instead of `ping`):

```
$ ansible all -m win_ping
win10.example.org | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

15.5 Windows Remote Management Troubleshooting

If you encounter unexpected problems in getting WinRM to work with Ansible, there are a few points you should check. Most problems fall into one of the following two categories:

- Network or connection problems
- Authentication problems

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

To resolve network issues, you can analyze the following points:

- Is the WinRM service running at all? (For example, you can use `Get-Service -Name winrm` in a PowerShell to check the status.)
- WinRM is a web service, so can you connect from the control host to the desired target by using a tool like `netcat`?
- Is a firewall blocking your access?
- Are there proxy settings like `http_proxy` or `https_proxy` that could be misdirecting your access?

To resolve authentication problems, you can proceed as follows:

- First, rule out typos in the parameterization, especially with `ansible_user` and `ansible_password`.
- Ensure that the target account has administrative permissions.
- If you use basic authentication, you should make sure to use a local account and not a domain account.
- If you use Kerberos (see the next section), you should pay attention to the capitalization of the realm name. The use of `EXAMPLE.ORG` in the example files is intentional!
- If you're using Kerberos, you should ensure that all systems involved are synchronized to the same time. Kerberos is extremely sensitive to even a few minutes of time difference.

15.6 Setup with an Active Directory and Kerberos

If your target systems are members of an active directory domain, you may want to use a domain account for Ansible, which will inherently involve Kerberos. Here's a step-by-step guide that you can easily adapt to your environment.

For this example, we assume that you can reach a domain controller for the `example.org` domain at the `192.168.150.70` IP address. The `win10.example.org` machine we already know is a domain member, and you should continue to administer it with Ansible.

Next, you proceed step by step as follows:

1. Change the nameserver to be used.

   Our control host does not need to become a domain member, but it should use the domain's domain name system (DNS) server. In the "vagrant lab," an entry in the */etc/resolv.conf* file is unfortunately not very long lasting because a dynamic host configuration protocol (DHCP) client is running there. Instead, the permanent solution is an entry in the */etc/dhcp/dhclient.conf* file:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
[...]
supersede domain-name-servers 192.168.150.70;
[...]
```

Listing 15.3: "/etc/dhcp/dhclient.conf"

If you make a subsequent call to `dhclient`, the setting should appear in the *resolv.conf*. In any case, you should now be able to resolve hostnames from the active directory (AD) domain!

2. Install the required Kerberos packages:

```
$ sudo apt install krb5-user python3-kerberos
```

The Debian installer may ask you for the Kerberos realm, and you should enter `EXAMPLE.ORG` there. However, we'll replace the configuration file shortly anyway.

3. Create the Kerberos configuration file:

You should adapt the */etc/krb5.conf* file according to . (The files delivered by default are mostly full of "nonsense."):

```
[libdefaults]
        default_realm = EXAMPLE.ORG

[realms]
        EXAMPLE.ORG = {
                kdc = dc1.example.org
        }

[domain_realm]
        .example.org = EXAMPLE.ORG
```

Listing 15.4: "/etc/krb5.conf": Example Kerberos Configuration File

4. Test the Kerberos setup.

To do this, you obtain a ticket for a domain account as a test:

```
$ kinit <username>@EXAMPLE.ORG
```

Other useful tools in this context are `klist` for listing all tickets and `kdestroy` for discarding tickets.

Now, you just need to update the inventory on our control host, as in Listing 15.3:

```
[windows]
win10.example.org

[windows:vars]
ansible_user=Administrator@EXAMPLE.ORG
ansible_password=PasswOrd
ansible_connection=winrm
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
ansible_winrm_transport=kerberos
ansible_winrm_server_cert_validation=ignore
```

Listing 15.3 "inventory": Example Windows Inventory

If you don't want the plaintext password in the file, you should remove the `ansible_password` line and instead add the following:

```
ansible_winrm_kinit_mode=manual
```

This makes Ansible rely on you to provide a suitable ticket with `kinit` before doing things like calling a playbook.

In any case, working with a domain member should now operate just as smoothly as with a standalone system:

```
$ ansible all -m win_ping
win10.example.org | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

15.7 Windows Modules

The Windows modules are now primarily distributed across the collections `ansible.windows` and `community.windows`, so I suggest that you take some time and skim through the introductory pages:

- *https://docs.ansible.com/ansible/latest/collections/ansible/windows/*
- *https://docs.ansible.com/ansible/latest/collections/community/windows/*

Of course, naming the top five or ten most important modules can be challenging because, much like in the Linux world, it really depends on *your* current needs.

You might be pleased to know that well-known standard modules such as `copy`, `template`, `file`, `get_url`, `user`, `group`, `lineinfile`, `service`, `command`, `shell`, `reboot`, and several others all have a `win_*` counterpart, which will make you feel right at home.

However, you should exercise caution when specifying Windows file system paths. The necessary backslash (\) in the Linux world often signifies an escape character, which can indeed cause problems. In a playbook, specifically in YAML, the backslash functions as an escape character only when it's in double quotation marks. Thus, the following usages are correct:

```
dest:  C:\Windows\Temp\test.txt
        dest: 'C:\Windows\Temp\test.txt'
        dest: "C:\\Windows\\Temp\\test.txt"
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

I strongly advise against using all other combinations, just as I advise against using the classic `key=value` form, where different rules apply. (You would always need two backslashes, to put it simply.)

It can get really bad in ad hoc mode, where the rules of the Linux command line or shell also apply:

```
$ ansible all -m win_copy \
    -a "dest=C:\\\\Windows\\\\Temp\\\\test.txt \
     content='Just a test'"
```

But to avoid making your life unnecessarily difficult, you should avoid such excursions.

I would now like to highlight some *really* Windows-specific modules.

15.7.1 "chocolatey.chocolatey.win_chocolatey": Package Management with Chocolatey

The Ansible modules for the popular Chocolatey package manager are now housed in their own collection called `chocolatey.chocolatey`.

```
- name: Install Notepad++ and PuTTY
  win_chocolatey:
    name:
      - notepadplusplus
      - putty.install
```

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/chocolatey/chocolatey/win_chocolatey_module.html*.

By the way, there's also a module for managing classic `.msi` or `.exe` installers. It's called `ansible.windows.win_package`.

15.7.2 "win_domain_user": Managing Domain Accounts

```
- name: Create user
  win_domain_user:
    name: jdoe
    firstname: John
    surname:   Doe
    password:  Passw0rd2024
```

Calls to this module must, of course, be directed to a domain controller.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/community/windows/win_domain_user_module.html*.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

15.7.3 "win_environment": Managing Environment Variables

```
- name: Set environment variable for all users
  win_environment:
    name: FAVORITE_ANIMAL
    value: Dog
    level: machine
```

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/windows/win_environment_module.html*.

15.7.4 "win_regedit": Managing Registry Entries

```
- name: Start every CMD.EXE with codepage 1252
  win_regedit:
    path: HKLM:\Software\Microsoft\Command Processor
    name: Autorun
    data: chcp 1252
```

To merely gather information about registry entries, use `win_reg_stat`.

The online documentation for this is at *https://docs.ansible.com/ansible/latest/collections/ansible/windows/win_regedit_module.html*.

15.7.5 "win_updates": Applying Updates

```
- name: Install important updates
  win_updates:
    category_names:
      - SecurityUpdates
      - CriticalUpdates
```

This module provides numerous additional features. If you need more information, please refer to the full documentation at *https://docs.ansible.com/ansible/latest/collections/ansible/windows/win_updates_module.html*.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 16
Callback Plug-ins

When Ansible processes a playbook, many things happen: a play begins, a task is successfully completed, another task fails, and much more. You'll usually see Ansible's reactions to all these *events* live in your terminal's output. The interesting news is that none of these are set in stone: through a plug-in mechanism, you can influence these reactions—up to self-programmed callback routines that can then do really anything.

There are three types of so-called *callback plug-ins*:

- **Stdout callback plug-ins**
  These control how the console output of `ansible-playbook` looks. *Only one* such output plug-in can be active at any given time.
- **Aggregate callback plug-ins**
  These plug-ins can output additional information (which is often statistical in nature) or write to files, (e.g., at the end of a task, role, or playbook).
- **Notification callback plug-ins**
  These allow information to be passed to other applications or services, such as databases, log servers, or communication services.

16.1 Stdout Callback Plug-ins

Table 16.1 shows a selection of the available output plug-ins. Plug-ins also have a namespace, which I include here for completeness.

Output Plug-ins	Descriptions
`ansible.builtin.default`	The usual output format
`ansible.builtin.minimal`	Output like the `ansible` command
`ansible.builtin.oneline`	Output like `ansible -o`
`ansible.posix.debug`	Detailed information in case of error
`ansible.posix.json`	Output entirely in JSON that's collected at the end of the playbook run
`ansible.posix.jsonl`	Like `ansible.posix.json` but per task

Table 16.1 Output Plug-ins

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Output Plug-ins	Descriptions
community.general.counter_enabled	Output with counters
community.general.dense	Extremely compact output
community.general.null	No screen output at all
community.general.selective	Only outputs tasks tagged with print_action
community.general.timestamp	Output with timestamps
community.general.unixy	UNIX-style output without much fuss
community.general.yaml	Output of data structures in YAML format

Table 16.1 Output Plug-ins (Cont.)

You can select such output plug-ins via the ANSIBLE_STDOUT_CALLBACK environment variable or with the Ansible configuration in the [defaults] section by using the stdout_callback directive. (Note that the ANSIBLE_STDOUT_CALLBACK environment variable is stronger.). Selecting via environment variable is certainly the easier way to perform testing or temporary activation; you can test it with the unixy plug-in, for example:

```
$ ANSIBLE_STDOUT_CALLBACK=unixy ansible-playbook […]
```

If you find a plug-in useful, you should also take a look at its official documentation at *https://docs.ansible.com/ansible/latest/collections/callback_index_stdout.html*.

The individual plug-ins often offer a surprisingly large number of configuration options; consider as an example the options of the default plug-in. (This is a directive with its default setting, along with the corresponding environment variable that can also control the setting.):

- **[defaults]/display_failed_stderr = false, ANSIBLE_DISPLAY_FAILED_STDERR**
  This option controls whether failed or unreachable tasks are written to STDERR.
- **[defaults]/display_ok_hosts = true, ANSIBLE_DISPLAY_OK_HOSTS**
  This option checks whether ok tasks or hosts are displayed.
- **[defaults]/display_skipped_hosts = true, ANSIBLE_DISPLAY_SKIPPED_HOSTS**
  This option checks whether skipped tasks or hosts are displayed.
- **[defaults]/show_custom_stats = false, ANSIBLE_SHOW_CUSTOM_STATS**
  This option checks whether statistical data that's collected with the set_stats module is displayed at the end.
- **[defaults]/show_per_host_start = false, ANSIBLE_SHOW_PER_HOST_START**
  This option checks whether an additional output line is generated for each host when a task starts.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

- **[defaults]/show_task_path_on_failure = false, ANSIBLE_SHOW_TASK_PATH_ON_FAILURE**
  This option checks whether the filename and line number of the failed task are displayed in case of an error.

16.2 Aggregate and Notification Callback Plug-ins

As already mentioned, there are a whole range of other callback plug-ins that are not (primarily) related to screen output. I just want to give you a quick overview of them here; for detailed information, I refer you to the official Ansible documentation at *https://docs.ansible.com/ansible/latest/collections/index_callback.html*.

First, you must activate all these plug-ins in the *ansible.cfg* by using the `callbacks_enabled` directive or the `ANSIBLE_CALLBACKS_ENABLED` environment variable if needed. You can also specify multiple plug-ins and separate them with commas. For example, you can activate the `timer` and `profile_tasks` plug-ins as follows:

```
# [defaults]
callbacks_enabled = timer,profile_tasks
```

Listing 16.1 "ansible.cfg": Activating Callback Plug-ins

Warning

Previously, this directive was called `callback_whitelist` (or `ANSIBLE_CALLBACK_WHITELIST`). However, starting with Ansible version 2.15, this designation is no longer supported.

The current name was introduced in version 2.11.

Many of these plug-ins require additional Python modules to function on the control host. This is also noted in the online documentation under the **Requirements** section, if applicable.

Table 16.2 and Table 16.3 show some of the available plug-ins.

Plug-ins	Descriptions
`ansible.builtin.junit`	It generates a JUnit file from the Playbook run.
`ansible.posix.cgroup_perf_recap`	It determines system activities using control groups (cgroups).
`ansible.posix.profile_roles`	It measures the time required for roles.
`ansible.posix.profile_tasks`	It measures the time required for tasks.

Table 16.2 Aggregate Callback Plug-ins

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Plug-ins	Descriptions
ansible.posix.timer	It adds the execution duration to the play overview.
community.general.cgroup_memory_recap	It determines memory usage by using cgroups.

Table 16.2 Aggregate Callback Plug-ins (Cont.)

Plug-ins	Descriptions
community.general.log_plays	It generates log files per host.
community.general.logdna	It provides LogDNA integration.
community.general.logentries	It provides Logentries integration.
community.general.logstash	It provides Logstash integration.
community.general.mail	It reports errors via mail.
community.general.nrdp	It provides integration with Nagios via the Nagios Remote Data Processor (NRDP).
community.general.slack	It provides Slack integration.
community.general.splunk	It provides Splunk integration.
community.general.sumologic	It provides Sumo Logic integration.
community.general.syslog_json	It provides system logging protocol (syslog) integration.
community.grafana.grafana_annotations	It provides Grafana integration.

Table 16.3 Notification Callback Plug-ins

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 17
Creating Your Own Collections and Modules

Collections have been the standardized distribution format for Ansible code (i.e., for playbooks, roles, modules, and plug-ins) since Ansible version 2.10. If you're genuinely interested in developing reusable components, you should definitely use Ansible version 2.10 or later. (Even in version 2.10, there are still some minor limitations, which I will discuss.)

For us, collections are not end goals; they serve primarily as containers. This chapter emphasizes self-developed content, particularly the modules. By now, you are familiar with developing playbooks and roles, so the next step is to ensure that they are appropriately "packaged."

Note

You can develop modules and plug-ins just like playbooks and roles, outside and independently of collections—the Ansible community has been doing this for years. However, given that the collection technique is now available, has proven effective, and addresses nearly all issues related to distribution and reuse, I see no reason to abandon it.

If you choose not to use collections as a "transport medium," then most of what we say in Section 17.4 and Section 17.5 about developing modules and plug-ins will still be correct and important. For information on distribution and usage, I refer you to the internet.

17.1 Namespaces, Names, and Setting Up a Collection Project

A new collection needs a name, and to avoid name collisions, it must be in a namespace. Both are simple identifiers formed from the allowed character set [`a-zA-Z0-9_`]. The namespace and the collection name are separated by a dot (.), and specific objects within the collection are noted after another dot.

So, if you want to use a `role1` role in a `collection1` collection in the `namespace1` namespace, the fully qualified collection name (FQCN) for this role is as follows:

```
namespace1.collection1.role1
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

For our further considerations regarding the creation of our own collections, we need suitable names. It appears that there aren't any specific reserved namespaces for private projects, so I'll suggest using `local` for now. A sample collection within this namespace could be simply named `demo`.

Furthermore, from a developer's perspective, we need a project directory for collections. We already discussed this in Chapter 8, Section 8.6, and I would like to stick to the approach we suggested there:

```
$ mkdir -p ~/ansible/collections/ansible_collections && cd $_
```

Important

The technically redundant intermediate *ansible_collections* directory is unfortunately necessary!

Now, you can create the basic layout for our new collection in this folder:

```
$ ansible-galaxy collection init local.demo
- Collection local.demo was created successfully
```

17.1.1 Ansible Project for Manually Testing the Collection

As you gradually populate our new collection, you must test these new components from a user's perspective. For this, I definitely recommend that you to start a new Ansible project, which you can quickly set up by using my script from Chapter 19, Section 19.1.2:

```
$ init-ansible-project.sh ~/ansible/projects/using_collections && cd $_
```

To ensure that you find our collection from this project, you should extend the collection search path via the project's own *ansible.cfg*:

```
# [default]
collections_path = ~/ansible/collections
```

Listing 17.1 "ansible.cfg": Extending Collection Search Path

Then, you can perform this test:

```
$ ansible-galaxy collection list local.demo
# /home/vagrant/ansible/collections/ansible_collections
Collection Version
---------- -------
local.demo 1.0.0
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The version number is derived from the *galaxy.yml* collection file, which also includes a wealth of well-annotated meta-information. We'll revisit this later.

17.1.2 Where Is Work Currently Being Done?

During development, you'll often need to switch between the collection and the test project. I advise you to open two separate windows or SSH sessions. To minimize confusion, I'll start each explanation with boldface **In the collection** or **In the test project** text whenever possible. The appropriate working directories for each situation are assumed implicitly.

In the collection:

```
$ cd ~/ansible/collections/ansible_collections/local/demo
```

In the test project:

```
$ cd ~/ansible/projects/using_collections
```

17.2 Playbooks in Collections

Let's start with the simplest case: a playbook. The logical place for this in a collection is the *playbooks* directory, which we still need to create.

In the collection:

```
$ mkdir playbooks
```

You can insert any simple playbook, like this:

```
---
- hosts: localhost

  tasks:
    - debug: msg="Hello from a collection-playbook!"
```

Listing 17.2 "playbooks/hello.yml": Collection Playbook

Important Difference from Normal Playbooks

The same rules for namespaces or collection names apply to the names of collection playbooks. In particular, no hyphen (-) is allowed!

In the test project, after brief hesitation, you might try the following:

```
$ ansible-playbook local.demo.hello
$ ansible-playbook local.demo.hello.yml
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

That's all perfectly correct, except that if you're still using Ansible version 2.10, you would only have the option of specifying the Linux path (i.e., collections wouldn't bring that many advantages at that point).

17.3 Roles in Collections

You can typically migrate existing roles into a collection with relative ease, provided that you adhere to the naming conventions for collection objects. Let's create a new role as an example.

In the collection, roles are located in the *roles/* directory. You need to create this folder, and in doing so, you will also generate a structure for a very simple role named `hello`:

```
$ mkdir -p roles/hello/tasks
```

For testing purposes, it's enough if the role includes just one task:

```
---
- debug: msg="Hello from a collection-role!"
```

Listing 17.3 "roles/hello/tasks/main.yml": Task File of Collection Role

In the test project, the following straightforward playbook is good enough for testing:

```
---
- hosts: localhost

  roles:
    - name: local.demo.hello
```

Listing 17.4 "use_role.yml": Playbook for Testing Collection Role

Alternatively, instead of using fully qualified names, you can list all namespaces on the playbook or role level with the `collections` keyword where Ansible should search for partially qualified objects. Please only make use of this feature if you consider the risk of a name collision to be very manageable. Specifically, it would look like this:

```
---
- hosts: localhost

  collections:
    - local.demo

  roles:
    - name: hello
```

Listing 17.5 "use_role2.yml": Using Search Paths and Partially Qualified Role Names

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

17.4 Modules in Collections

In this section, I aim to demonstrate that Ansible modules are straightforward and that you can easily develop your own if necessary. Common reasons for creating your own modules include the following:

- You want to perform a manageable action on target systems, but none of the existing modules seem suitable for this.
- Using the `command` or `shell` module is too complicated or error-prone for the task.
- Using the `script` module seems boring and somehow not in line with Ansible.
- A role is too cumbersome or unsuitable for the purpose.

This section is primarily intended for programmers. While in theory, you can create modules in any programming language, you should be proficient in at least one language to explore the subject in depth.

The Python programming language enjoys special privileges here. Since Ansible itself is based on Python, the developers have provided an extensive library interface that greatly simplifies module creation and offers slight performance advantages during execution. However, I would like to first illustrate the principles of module creation with the simplest possible examples in the Bash language, because Bash knowledge is certainly more widespread among Ansible users than Python knowledge.

17.4.1 Getting Started

First, we need to clarify the storage location for modules within a collection, as it might not align with your initial expectations.

In the collection, modules are stored in the *plugins/modules/* folder:

```
$ mkdir -p plugins/modules
```

As mentioned earlier, Ansible modules are just programs. There is one requirement we need to address from the start: the module (= program) is expected to produce output in JSON format.

For a first test, we'll create the simplest module possible by using the Bash programming language:

```
#!/bin/bash
echo '{}'
```

Listing 17.6 "plugins/modules/hello.sh": Example First Module in Bash Language

The program undoubtedly outputs valid JSON, so everything should be fine. If you only want to call it via Ansible, you don't even need to grant execution rights. For simplicity's sake, you should test it **in the test project** with an ad hoc call:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
$ ansible localhost -m local.demo.hello
localhost | SUCCESS => {
    "changed": false
}
```

Note that you *must* omit the file extension (which in our case is *.sh*).

Typically, the returned JSON structure includes several entries by default. You can include any data with it, but there are only three keys with predefined meanings:

- **changed**
  With `changed`, you indicate whether the execution of the module has caused a state change of the host. You should always provide this value (otherwise, `false` will be assumed).
- **failed**
  In case of an error, you use this entry to signal that something went wrong (`"failed": true`).
- **msg**
  You can provide a helpful message with `msg` in case of an error.

The `random_state` module in Listing 17.7 simulates all of these events randomly:

```
#!/bin/bash

NUMBER=$(( RANDOM % 3 ))

# 0 = no Change
# 1 = Change
# 2 = Error

case $NUMBER in
    0) JSON='"changed": false'
       ;;
    1) JSON='"changed": true'
       ;;
    2) JSON='"changed": false, "failed": true, "msg": "Something went wrong"'
       ;;
esac

cat <<EOF
{
  "random_number": $NUMBER,
  $JSON
}
EOF
```

Listing 17.7 "plugins/modules/random_state.sh": Example Module in Bash Language

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

As luck would have it, you might have to call the module many times to observe all three possible outcomes:

```
$ ansible localhost -m local.demo.random_state
localhost | SUCCESS => {
    "changed": false,
    "random_number": 0
}
$ ansible localhost -m local.demo.random_state
localhost | CHANGED => {
    "changed": true,
    "random_number": 1
}
$ ansible localhost -m local.demo.random_state
localhost | FAILED! => {
    "changed": false,
    "msg": "Something went wrong",
    "random_number": 2
}
```

17.4.2 Module Parameters

As we know, most modules need specific parameters to function; this is what makes them flexible and useful. Custom modules will undoubtedly be no exception.

Let's create an example module called `calculate`, which combines two integers with one of the four basic arithmetic operations.

In the test project, the application in a playbook could look like this:

```
- hosts: localhost

  tasks:
    - name: Calculate the sum of two numbers
      local.demo.calculate:
        number1: 3
        number2: 4
        operator: "+"
      register: calc

    - debug:
        msg: The result is {{calc.result}}
```

Listing 17.8 "calc_test.yml": Example Playbook for Testing "calculate" Module

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

The specification of the `number1` and `number2` parameters should be mandatory, and the `operator` could have the default `+`.

In the module, you naturally want to access these parameters in some way. To achieve this, you must understand what Ansible does in the background when a module is invoked:

1. Ansible generates a temporary *parameter file*, which contains a long line in the following form:

   ```
   number1=77 number2=99 operator=+
   ```

   (Ansible adds more internal key-value pairs, but we don't need to worry about those at the moment.)
2. Ansible transfers the parameter file along with the module to the target system.
3. On the target system, Ansible determines the interpreter from the shebang line and initiates a call in the form:

   ```
   <INTERPRETER> <MODULE> <PARAMETER FILE>
   ```

From the perspective of a shell script, the parameter file (thanks to the matching format) only needs to be read in with the `source` command. (We'll leave security concerns aside for now.) This way, all parameters are available as shell variables with the same name.

In the collection, a solution could look something like this:

```
#!/bin/bash

# Load module parameters:
source $1

# In case of missing operator we choose "+":
operator=${operator:-+}

# Bash does the job:
result=$(( $number1 $operator $number2 ))

if [ $? -eq 0 ]; then
    JSON='"result": "'$result'"'
else
    # Something went wrong
    JSON='"msg": "Syntax error", "failed": true'
fi

# Generate JSON output:
cat <<EOF
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
{
  "changed": false,
  $JSON
}
EOF
```

Listing 17.9 "plugins/modules/calculate.sh": Module with Parameter Processing

In the test project, you can test it with the playbook or ad hoc:

```
$ ansible localhost -m local.demo.calculate -a "number1=3 number2=4 operator=+"
localhost | SUCCESS => {
    "changed": false,
    "result": "7"
}

$ ansible localhost -m local.demo.calculate -a "number1=44 number2=55"
localhost | SUCCESS => {
    "changed": false,
    "result": "99"
}

$ ansible localhost -m local.demo.calculate -a "number1=6 number2=7 operator=*"
localhost | SUCCESS => {
    "changed": false,
    "result": "42"
}

$ ansible localhost -m local.demo.calculate -a "number1=200"
localhost | FAILED! => {
    "changed": false,
    "msg": "Syntax error"
}
```

Of course, the error messages could be more detailed ones like `Missing number` or `Unknown operator`, but I didn't want to stray too far from the essentials.

As you've already learned, parameter files are generated in the `Key=Value` format. For a shell-based module, this is extremely practical—a single `source` call is enough, and all parameters are available for internal processing.

However, other programming languages usually cannot process this format as easily, and I don't recommend that you develop your own parser. As an alternative, you have the option to obtain the parameter file in JSON format. To do this, you simply need to "wish" for it by inserting a comment line in the following form somewhere in your script:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
# WANT_JSON
```

This will prompt Ansible to create the parameter file in JSON format, as shown here:

```
{"number1": 77, "number2": 99, "operator": "+"}
```

A *shell* program can't immediately benefit from this format; in this case, you would probably rely on the external `jq` program for parsing, which, naturally, must then be available on all target hosts.

17.4.3 Modules with Python: Example Problem Statement

To demonstrate module development in Python, we want to tackle a more exciting task. The module to be developed, `testfiles`, should create a certain number of files with a specific sequential naming scheme in a selectable folder. So, it's something like what you would accomplish in the Shell with a simple command like this:

```
Create 20 files test1.txt to test20.txt in /tmp:
$ cd /tmp; seq -f test%g.txt 20 | xargs touch
```

The new module is essentially a gimmick, but it will allow me to demonstrate all relevant aspects such as parameter processing, idempotency behavior, check mode, and more. This way, I can prepare you to launch your own "serious" project afterward.

A hypothetical playbook to test the new module looks like this:

```
- hosts: localhost

  tasks:
    - name: Create 20 testfiles
      local.demo.testfiles:
        path: /tmp
        number: 20
        format: "test%g.txt"
        state: present
```

Listing 17.10 "pymodule_test.yml": Playbook for Testing "testfiles" Module

We'll agree on the following meanings or uses of the four possible parameters:

- **`path`**
  This is the target directory, and the default is */tmp*. The path must already exist; otherwise, it will result in an error.
- **`number`**
  This is the number of test files, and the default is `10`.
- **`format`**
  This is a required parameter that specifies the naming scheme in `printf` format.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

- **state**
  Possible options are `present` (the default) and `absent`.

17.4.4 An Example Solution

In Listing 17.11, you can see a solution for the given problem, which you should store **in the collection**. I will first present it as a whole and then break it down and explain it:

```
#!/usr/bin/python

from ansible.module_utils.basic import AnsibleModule
import os

def run_module():

    module_args = dict(
        path  =dict(type='path', required=False, default='/tmp'),
        number=dict(type='int',  required=False, default=10),
        format=dict(type='str',  required=True),
        state =dict(choices=['present', 'absent'],
                    required=False,
                    default='present'
        )
    )

    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    path  = module.params['path']
    state = module.params['state']

    if not os.path.isdir(path):
        module.fail_json(msg=path + ' is not an existing directory')

    result = dict(changed=False)

    for i in range(1, module.params['number'] + 1):
        filepath = path + '/' + ( module.params['format'] % (i) )

        if state == 'present' and not os.path.exists(filepath):
            if not module.check_mode:
                os.mknod(filepath)
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
            result['changed'] = True

        if state == 'absent' and os.path.exists(filepath):
            if not module.check_mode:
                os.remove(filepath)
            result['changed'] = True

    module.exit_json(**result)


def main():
    run_module()

if __name__ == '__main__':
    main()
```

Listing 17.11 "plugins/modules/testfiles.py": Example Module in Python Language

In the test project, you can test it with the playbook or ad hoc:

```
$ ansible localhost -m local.demo.testfiles -a "format='test%02g.txt'"
localhost | CHANGED => {
    "changed": true
}

Here's the same again (in an idempotency test):
$ ansible localhost -m local.demo.testfiles -a "format='test%02g.txt'"
localhost | SUCCESS => {
    "changed": false
}

Here's a check:
$ (cd /tmp; ls *.txt)
test01.txt  test03.txt  test05.txt  test07.txt  test09.txt
test02.txt  test04.txt  test06.txt  test08.txt  test10.txt

And then, you delete it:
$ ansible localhost -m local.demo.testfiles -a "format='test%02g.txt' \
    state=absent"
localhost | CHANGED => {
    "changed": true
}
```

You should feel free to make further calls with other, missing, or invalid parameters. The module should respond reasonably well.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

17.4.5 Explanations and Further Possibilities

Let's go through the solution step by step:

```
from ansible.module_utils.basic import AnsibleModule
import os
```

The `AnsibleModule` class provides the necessary functionality, and `os` is a standard Python module for file management. Both are imported here.

```
def run_module():

    module_args = dict(
        path  =dict(type='path', required=False, default='/tmp'),
        number=dict(type='int',  required=False, default=10),
        format=dict(type='str',  required=True),
        state =dict(choices=['present', 'absent'],
                    required=False,
                    default='present'
        )
    )
```

The `run_module()` function is the core of our module. The name is arbitrary, as it is just called at the end of the program.

We will need the `module_args` dictionary (or map) as an argument to a constructor call in the next line. It includes a detailed description of all module parameters, which are the keys on the left side. Each value is also a dictionary that defines the properties of the respective parameter:

- **`type`**
  The `type` refers to the data type of the parameter. By default, everything would be text, but by setting it, Python can convert directly into the appropriate internal data type. The basic types available are `str`, `path`, `bool`, `int`, and `float`. Composite types like `list` and `dict` are also possible, and you can specify them by the comma-separated user (`month=April,May,June` or `animals=Dog=woof,Cat=meow`).
- **`required`**
  This indicates whether it's a mandatory parameter. The default is `false`, but we recommend that you always explicitly mention this attribute.
- **`default`**
  This specifies a default value that is used if the user does not set the parameter. In our case, you could certainly debate whether a default of ten files makes sense or whether it would have been better to make it a mandatory parameter.
- **`choices`**
  This defines a set of possible values (a selection list).

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

- **aliases**
  This is a list of alternative names for a parameter. We don't use it in our example, but you might recognize it from other modules like `lineinfile`. (Look for "aliases" in the documentation there.)

Now, back to the action:

```
module = AnsibleModule(
    argument_spec=module_args,
    supports_check_mode=True
)
```

Here, we construct an instance of the `AnsibleModule` class, in which, among other things, the parameters are read and checked against our specification.

The `supports_check_mode` setting indicates whether the module is capable of operating *meaningfully* in check mode (with the `--check` option). In this case, no system changes should be made; the module should only simulate them. If a module doesn't support check mode, it will simply be skipped when the playbook containing it is executed.

```
path  = module.params['path']
state = module.params['state']
```

The user-provided parameter values are now accessible. To simplify access for some applications, two copies have been created.

```
if not os.path.isdir(path):
    module.fail_json(msg=path + ' is not an existing directory')
```

According to our requirements, the target directory must exist. If it doesn't, we'll terminate the process with an error. The `fail_json` method takes care of the tedious details and handles everything required.

```
result = dict(changed=False)

for i in range(1, module.params['number'] + 1):
```

We'll return the `result` dictionary at the end. No system changes have been made yet, but that could still change. To create the files, you start a loop from 1 to the desired number. The Python `range()` function returns `N-1` as the last value, hence the `+1`:

```
    filepath = path + '/' + ( module.params['format'] % (i) )
```

The name of the file to be created is generated here. With the `%` operator, you can insert the parameters on the right side into a classic `printf` format string.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    if state == 'present' and not os.path.exists(filepath):
        if not module.check_mode:
            os.mknod(filepath)
        result['changed'] = True

    if state == 'absent' and os.path.exists(filepath):
        if not module.check_mode:
            os.remove(filepath)
        result['changed'] = True
```

This is the program logic that handles the creation or deletion of test files. In check mode, it only pretends to do so. If a change to the target system is necessary, this is recorded in the `result` dictionary.

```
    module.exit_json(**result)
```

This ends the module and returns the required JSON structure, which is generated from the `result` dictionary. The double asterisk (**) is a Python feature; it breaks down the dictionary into individual key-value parameters as required by the `exit_json` method.

The rest of the program consists only of Python-specific indirections, and that lets you use the Python module as both a module and a standalone program.

17.4.6 Embedded Documentation

Our module primarily needs standardized documentation. Adding some comments in the source code would be beneficial, and we could also consider embedding specific YAML sections directly within the code, which can then be read by tools like `ansible-doc`. You would just add a `DOCUMENTATION` variable and an `EXAMPLES` variable outside of any functions containing YAML-formatted text, with a special structure as their value:

```
# [...]

DOCUMENTATION = '''
---
module: testfiles
short_description: Create a number of of testfiles
description:
  - This module can create or remove a number of testfiles.
  - Number, name format, and destination path can be chosen.
version_added: "1.0.0"
options:
  path:
    description:
      - The destination folder.
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
        It must exist; otherwise, this module will fail.
    type: path
    default: /tmp
  number:
    description:
      - Number of files
    type: int
    default: 10
  format:
    description:
      - Format in printf-style (like 'test%02g.txt')
    type: str
    required: true
  state:
    description:
      - Whether the files should be created or removed.
    type: str
    choices: [ absent, present ]
    default: present

author:
  - Some User (user1@example.org)
'''

EXAMPLES = '''
- name: Create 20 testfiles test01.txt .. test20.txt in /tmp
  testfiles:
    number: 20
    format: test%02g.txt

- name: Remove 30 testfiles test01.jpg .. test30.jpg in /opt
  testfiles:
    number: 30
    format: test%02g.jpg
    path: /opt
    state: absent
'''
```

Listing 17.12 "plugins/modules/testfiles.py": Example Inline Documentation of Module

You can then use `ansible-doc` to see how this information is utilized:

```
$ ansible-doc local.demo.testfiles
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

17.4.7 Outlook

Even after all you've learned about developing your own modules, there's still much to explore. However, I will wrap up this topic here to keep us aligned with the book's main focus.

If you want to delve a little deeper, you should start with the official documentation at *https://docs.ansible.com/ansible/devel/dev_guide/developing_modules_general.html*. And if you encounter a specific problem and don't know how to proceed, then whenever possible, you should look at the source code of a module that does something similar. Take advantage of open source!

17.5 Plug-ins in Collections

Before I guide you to creating an example custom plug-in, let me briefly clarify the difference between modules and plug-ins:

- **Modules** are standalone programs that you can invoke via the Ansible interface (typically, via the `ansible` or `ansible-playbook` command). They are executed on target hosts, which are usually remote machines. In principle, you can create modules in any programming language.
- **Plug-ins** extend the functions of the Ansible core and are executed exclusively on the control host. This modularity allows you to adjust or change the behavior of Ansible in many areas (e.g., when establishing connections to target systems [with connection plug-ins] or when processing certain events [with callback plug-ins]). You can only implement plug-ins with Python.

17.5.1 An Example Callback Plug-in

You've already learned that there are different types of plug-ins in Ansible. At this stage, it's not feasible for us to explore all of them in terms of custom development, so for simplicity's sake, I'll focus on an example output plug-in.

If you want to try it, you can create the *plugins/callback/* folder **in the collection**:

```
$ mkdir plugins/callback
```

Then, you can place the following Python file from Listing 17.13 there:

```
from ansible.plugins.callback import CallbackBase

class CallbackModule(CallbackBase):

    CALLBACK_VERSION = 2.0
    CALLBACK_TYPE = 'stdout'
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    CALLBACK_NAME = 'testcallback'

    def v2_playbook_on_start(self, playbook):
        self._display.display("Starting playbook")

    def v2_playbook_on_play_start(self, play):
        self._display.display("Starting play")

    def v2_playbook_on_task_start(self, task, is_conditional):
        self._display.display("Task: " + task.get_name())

    def v2_playbook_on_stats(self, stats):
        self._display.display("Finished")

    def v2_runner_on_ok(self, result):
        self._display.display("OK: " + result._host.get_name())

    def v2_runner_on_skipped(self, result):
        self._display.display("SKIPPED: " + result._host.get_name())
```

Listing 17.13 "plugins/callback/testcallback.py": Example Callback Plug-in

In the test project, you can try it with any playbook, and make sure to provide a suitable inventory if necessary. I'm using an example from our neighboring `start` project here:

```
$ ANSIBLE_STDOUT_CALLBACK=local.demo.testcallback ansible-playbook \
    -i ../start/inventory \
       ../start/playbooks/apache2.yml
Starting playbook
Starting play
Task: Gathering Facts
OK: debian
OK: ubuntu
Task: Update package index
OK: debian
OK: ubuntu
Task: Install Apache package
OK: debian
OK: ubuntu
Task: Start service and enable start on boot
OK: debian
OK: ubuntu
Task: Upload minimal homepage
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
OK: debian
OK: ubuntu
Task: Upload config file
OK: debian
OK: ubuntu
Task: Activate config
OK: debian
OK: ubuntu
OK: debian
OK: ubuntu
Finished
```

Here, you can see how Ansible invokes custom callback routines for various events.

17.5.2 Outlook

Before you dive into creating more advanced plug-ins, you should explore the online resource at *https://docs.ansible.com/ansible/devel/dev_guide/developing_plugins.html*. There, you'll find code examples and tips for all kinds of callbacks.

17.6 Deploying and Installing Collections

We've already discussed many key points in Chapter 8, Section 8.6, that focused on the practical reusability of roles. My aim there was to provide a straightforward and practical solution, avoiding cross-references and broad considerations, and I apologize for any resulting redundancy on this topic in this section.

The basic idea is to manage collections with the `ansible-galaxy` command. The default source for collections is (you guessed it) Ansible Galaxy (*https://galaxy.ansible.com*). And as great and important as this online repository is, I boldly claim that most of my esteemed readers do not want to deploy their roles or collections (at least initially) in Galaxy.

Note

If you do plan to publish in Galaxy—great! However, you must first deal with certain quality criteria and metadata formats like the *galaxy.yml* file. For this, you should use the official documentation at *https://docs.ansible.com/ansible/devel/dev_guide/developing_collections.html* as a starting point.

Some of the concepts I describe here can even be helpful to those who only want to maintain private collections, but you can easily work by the seat of your pants at first and still accomplish things correctly and effectively.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Let's tackle this matter pragmatically. You should start in the base folder of your self-developed `local.demo` collection, and you should adjust the *galaxy.xml* file in meaningful and relevant places and maybe write a few helpful words in the file *README.md*.

Then, you should turn it into a Git project (let's say `ansible/collection-local-demo`) and save everything to your designated Git server.

Installation

With the `ansible-galaxy` command, you can reinstall this collection locally by specifying a (somewhat strange) Git URL (assuming you have appropriate access rights on the Git server):

```
$ ansible-galaxy collection install \
    git+https://gitea.example.org/ansible/collection-local-demo.git
```

The default installation target for `collection install` is the *~/.ansible/collections/* folder, but with the `-p <COLLECTIONS_PATH>` option, you can enforce a different installation target.

Important

Unfortunately, the default installation target is also changed by project-specific settings (`ANSIBLE_COLLECTIONS_PATH` or `collections_path`). If you have even the slightest doubt, you should always work with the explicit `-p` option—especially when an unconditional overwrite is requested with `-f`!

For easier or automatable installation of required collections, a *requirements.yml* file is usually provided in the project:

```
---
collections:
  - name: https://gitea.example.org/ansible/collection-local-demo.git
    type: git
```

Listing 17.14 "requirements.yml": Listing of Required Collections

Now, you can install all the collections listed there with the following command:

```
$ ansible-galaxy collection install -r requirements.yml
```

Note

In a project, this file is usually located in the *collections/* folder because it's searched for and processed by web applications like AWX by default.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Finally, I'd like to demonstrate how to specify a collection from Ansible Galaxy in a *requirements* file using the correct syntax:

```
collections:
  - name: community.vmware
    type: galaxy
```

Listing 17.15 "requirements.yml": Listing of Galaxy Collection

You can entirely omit the type specification since, as I previously mentioned, Galaxy is the default source for collections.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 18
Developing and Testing with Molecule

Whatever you're currently doing with Ansible—unless it's just ad hoc commands—you are likely conducting some form of testing simultaneously. However, when a project demands exceptionally high quality, relying solely on manual testing is often inadequate because of the following:

- Code changes can have unpredictable side effects.
- You can't manually repeat all tests on every target platform after every change.

In the world of software development, there's a well-established solution: automated and consistently reproducible software tests. Why shouldn't that solution also be possible in the Ansible world? Ultimately, developing playbooks and roles is simply another form of software development—and *infrastructure as code* (IaC) is the magic word.

In recent years, the *Molecule* test framework has matured into the de facto standard for Ansible. The concept behind Molecule is to automate an entire testing cycle by following these steps:

1. Provisioning suitable virtual target hosts
2. Executing the role or playbook to be tested
3. Checking or validating whether everything is in the expected state afterward
4. Removing the target hosts without a trace

Molecule manages everything by using specifically crafted Ansible playbooks, and the entire system is highly flexible. For instance, in step 1, you can freely choose your technology (such as Docker, Podman, Vagrant, Elastic Compute Cloud (EC2), or Hetzner Cloud), and in step 3, you have the option to select from various validation systems (including Ansible, Testinfra, and Serverspec).

18.1 Preparations and Setup

In this section, I'd like to focus on the variant that involves local containers because it's by far the most prevalent variant. No other test environment can be set up and torn down as swiftly as a cluster of Docker or Podman containers.

Specifically, I want to use Podman as the container engine because it runs a bit more smoothly with Molecule than Docker. Also, you might already have Podman installed

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

(e.g., if you have worked through the chapter on web interfaces). Otherwise, quickly install the `podman` distribution package now.

The first major challenge is acquiring the necessary images for testing. In many tasks you aim to achieve with Ansible, this ultimately involves installing and starting a service, which requires an initialization system. (Typically, this would be *systemd*.) In a typical container image, however, this software is not available, so in most cases, we first need very specific images that meet this requirement.

If you don't want to build them yourself, I recommend that you use the images from Jeff Geerling (*https://hub.docker.com/u/geerlingguy*). Jeff is a leading figure in the Ansible community and the author of an Ansible book, and software released by him should be regarded as completely trustworthy. So, you should download one of his images as follows and try it out—and I would recommend downloading all the necessary images in advance anyway. (Molecule does this internally when needed, but since everything is controlled by Ansible, you wouldn't have a useful progress indicator if you just let Molecule do that):

```
$ podman pull docker.io/geerlingguy/docker-ubuntu2204-ansible
More images may be here:
$ podman pull docker.io/geerlingguy/[...]
```

Now, you should start a container for testing. The key question is whether the systemd environment runs error-free within the container:

```
$ podman run -d --name u1 docker.io/geerlingguy/docker-ubuntu2204-ansible

$ podman exec -it u1 bash

root@d8392c19095e:/# systemctl status
* d8392c19095e
    State: running <-- IMPORTANT!
[…]

root@bb9596c2b5f3:/# exit

$ podman rm -f u1
```

With Docker, you may need to add Sthe `run` parameters that are called `--privileged`, `-v /sys/fs/cgroup:/sys/fs/cgroup`, and `--cgroupns=host`. We are grateful that *Podman* seems to spare us from that.

You can install Molecule by using Python package management:

```
$ sudo pip3 install molecule molecule-plugins[podman]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

18.2 Getting Started

You can certainly integrate Molecule into existing roles, but let me first demonstrate how to incorporate it from the outset when developing something new. It's best to include the new role in a collection right from the start, so as an example, let's develop a role together that installs a PostgreSQL-17 database on an Ubuntu-22.04 system. This is particularly interesting because it involves integrating an external repository, and it's a refreshing change from always dealing with Apache.

Let's use the same directory structure as in Chapter 17, Section 17.1:

```
$ mkdir -p ~/ansible/collections/ansible_collections && cd $_

$ ansible-galaxy collection init local.database
- Collection local.database was created successfully
```

Now, you can continue with creating a role structure within the collection:

```
$ cd local/database/roles

$ ansible-galaxy role init postgresql
- Role postgresql was created successfully
```

Within the role, you can now initialize the Molecule directory hierarchy:

```
$ cd postgresql

$ molecule init scenario -d podman
```

This command adds another *molecule* folder to the role structure, which contains a *converge.yml* playbook and a *molecule.yml* configuration file in a so-called *scenario directory* named *default*. You will edit the last of these immediately because Molecule (as discussed in the previous section) will be working with very specific images:

```
---
driver:
  name: podman
platforms:
  - name: ubuntu2204
    hostname: ubuntu-2204
    image: geerlingguy/docker-ubuntu2204-ansible:latest
    command: ""
    pre_build_image: true
# Just defaults:
provisioner:
  name: ansible
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
verifier:
  name: ansible
```

Listing 18.1 "molecule/default/molecule.yml": Configuration File for Molecule

Note

If you want to test not just a single role but multiple collection components, you can also initialize Molecule at a higher directory level. The documentation recommends creating directory extensions in the root folder of the collection and then running molecule init inside it.

In any case, make sure that the *molecule/* is in the current working directory where you want to run `molecule`. In our case, this serves as the role's base folder, where you can get an overview by using this:

```
$ molecule list

Instance Name | Driver Name | Provisioner Name | Scenario Name | Created
--------------+-------------+------------------+---------------+---------
ubuntu2204    | podman      | ansible          | default       | false
```

(The output has been slightly condensed.)

You can create all required containers, which in our case is just one container for now, with this:

```
$ molecule create
```

Then, if everything goes well, you can connect to the container with this:

```
$ molecule login -h ubuntu2204
root@ubuntu-2204:/# exit
```

As long as you have only one container, using a simple `molecule login` is sufficient.

Now, let's take a look at how Molecule can assist in the development of a new role.

18.3 Develop

Molecule expects a test playbook named *converge.yml* in the scenario folder. You'll probably find one there already with a simple `debug` task. Then, you'll want to call our role, for which the following playbook is suitable:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- name: Converge
  hosts: all

  tasks:
    - name: Include postgresql role
      include_role:
        name: "local.database.postgresql"
```

Listing 18.2 "molecule/default/converge.yml": Test Playbook for Molecule

With the following command, you can instruct Molecule to start this playbook in the test environment (which will be provisioned beforehand if necessary):

```
$ molecule converge
```

Converge means "*come closer together*," and the goal is for the state of the test environment to align as closely as possible with the state outlined in the role. Since our role is currently inactive, nothing noteworthy will occur yet.

Now, let's bring the role to life. In the following *tasks/main.yml* file, we set the stage for potential future target platforms:

```
---
- name: Load platform-specific tasks
  include_tasks: "{{ ansible_distribution | lower }}.yml"
```

Listing 18.3 "tasks/main.yml": Entry Point into Role

You should feel free to run `molecule converge` again to test it. The run naturally will fail because the file to be included does not exist, and you need to fix this. You should refer to the following listing for suggestions:

```
---
- assert:
    that:
      ansible_distribution_version is version('22.04', '==')

- name: Update apt cache
  apt:
    update_cache: yes
    cache_valid_time: 3600

- name: Install some prereqs
  apt:
    name:
      - apt-transport-https
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
      - ca-certificates
      - gnupg

- name: Import the repository signing key
  get_url:
    url: https://www.postgresql.org/media/keys/ACCC4CF8.asc
    dest: /etc/apt/keyrings/pgdg-archive-keyring.asc

- name: Add PostgreSQL apt repo
  apt_repository:
    repo: >-
      deb [signed-by=/etc/apt/keyrings/pgdg-archive-keyring.asc]
      https://apt.postgresql.org/pub/repos/apt
      {{ansible_lsb.codename}}-pgdg main
    filename: pgdg
    state: present

- name: Install PostgreSQL 17
  package:
    name: postgresql-17
    state: present

- name: Start and enable service
  service:
    name: postgresql
    state: started
    enabled: true
```

Listing 18.4 "tasks/ubuntu.yml": Ubuntu-Specific Tasks for PostgreSQL Installation

With a quick `molecule converge` and a little patience, you can log in to the target host with `molecule login` and check whether everything is in order:

```
$ molecule converge
[…]

$ molecule login
root@ubuntu-2204:/# ps -ef | grep postgres
postgres  3946     1  [...] /usr/lib/postgresql/17/bin/postgres [...]
postgres  3948  3946  [...] postgres: 17/main: checkpointer
postgres  3949  3946  [...] postgres: 17/main: background writer
[...]
root@ubuntu-2204:/# exit
```

As you can see, we already have a very nice "development environment"—but we haven't really *automated testing* yet.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

18.4 Testing with the Ansible Verifier

The starting point for all tests with the Ansible verifier is a playbook named *verify.yml* in the scenario folder. After a `converge`, these tests check whether all expected goals have been achieved. As you might expect, you trigger this process with this:

```
$ molecule verify
```

Take a moment to try it; since there is no verify playbook available yet, nothing should fail.

Now, you may ask, "Which tests should we develop?" Let's go through the tasks of the role step by step to identify candidates:

- There is nothing to test in the `assert` task.
- In the `Update apt cache` task, no one would seriously doubt or feel the need to verify whether the apt cache was indeed updated.
- In the `Install some prereqs` task, it might seem a bit odd to verify whether the three packages were actually installed. While you can certainly do that, there are other tasks on your agenda.
- The `Import signing key` task is a better candidate for testing. After all, a lot could go wrong there.
- The `Add PostgreSQL apt repo` task is another good candidate for testing, but should we really test whether the `apt_repository` module is doing its job? In any case, we'll definitely write a test at least for educational reasons.
- You can probably leave the `Install` task untested; what matters most is whether the PostgreSQL service is running after the final task. There will certainly be a test for this.

In Listing 18.5, you can see a verify playbook that implements the corresponding tests:

```
---
- name: Verify
  hosts: all
  gather_facts: false

  vars:
    key_filename:  /etc/apt/keyrings/pgdg-archive-keyring.asc
    repo_filename: /etc/apt/sources.list.d/pgdg.list

  tasks:
    - name: Install file package
      package:
        name: file
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
  - name: Test | Check content of key file
    command: file {{key_filename}}
    changed_when: false
    register: file_cmd
    failed_when: >
      'PGP public key block Public-Key' not in file_cmd.stdout

  - name: Test | "{{repo_filename}} has correct content"
    lineinfile:
      path: "{{repo_filename}}"
      regexp: '^deb.*https://apt\.postgresql\.org/pub/repos/apt.*main$'
      state: absent
    check_mode: yes
    register: conf
    failed_when: conf is not changed

  - name: Test | PostgreSQL service is running
    wait_for:
      port: 5432
      timeout: 30
    register: port_check
    failed_when: port_check.failed
```

Listing 18.5 "molecule/default/verify.yml": Molecule Tests with Ansible Verifier

Now, you should start Molecule:

```
$ molecule verify
```

You'd be right to question whether a test procedure should involve installing a package, but primarily, I wanted to demonstrate a test using a command call (which is the first test task).

The technique with `check_mode`, `register` + `failed_when` in the second test task has a certain elegance, but you can also readily see that creating tests with Ansible syntax is not pure pleasure. In fact, you could invest three times as much time developing these tasks as in performing the tasks that do the actual job.

Note

The difficulty of writing tests should never be an excuse for not writing them. However, don't deceive yourself by only crafting "simple" tests for trivial matters while neglecting the truly challenging and interesting aspects.

Perhaps Ansible playbooks are not the optimal testing tool either? Thankfully, there are alternatives, as the following section shows.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

18.5 Testing with the Testinfra Verifier

If you happen to know a little Python, you might feel more comfortable with Python tests. Before using Testinfra, you need to install it, for instance by using a distribution package:

```
$ sudo apt install python3-testinfra
```

(If this version is too old for your taste, you can instead install the `pytest-testinfra` Python package via *pip3*.)

After that, you need to inform Molecule (in the *molecule.yml* configuration file) that you want to use an alternative verifier:

```
# [...]
verifier:
  name: testinfra
  options:
    connection: podman
```

Listing 18.6 "molecule/default/molecule.yml": Changing Verifier to Testinfra

The mechanics of the tests are as follows. First, you create a folder named *tests/* in the scenario directory:

```
$ mkdir molecule/default/tests
```

Then, you can add any number of Python files, and each name should start with `test_`. Listing 18.7 shows an example file that implements the tests from the previous section more or less analogously by using Python/Testinfra:

```
import testinfra.utils.ansible_runner
import os
import re

testinfra_hosts = testinfra.utils.ansible_runner.AnsibleRunner(
    os.environ['MOLECULE_INVENTORY_FILE']).get_hosts('all')


def test_key_file(host):
    key_filename = "/etc/apt/keyrings/pgdg-archive-keyring.asc"
    key_file = host.file(key_filename)
    assert key_file.exists

    command = host.run("apt install -y file")
    assert command.succeeded
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    command = host.run(f"file {key_filename}")
    assert 'PGP public key block Public-Key' in command.stdout


def test_sources_list_content(host):
    content = host.file("/etc/apt/sources.list.d/pgdg.list") \
        .content_string

    assert re.search(
        "^deb.*https://apt\\.postgresql\\.org/pub/repos/apt.*main",
        content
    )


def test_postgresql_is_running_and_enabled(host):
    pg = host.service("postgresql")
    assert pg.is_running
    assert pg.is_enabled
    assert host.socket('tcp://127.0.0.1:5432').is_listening
```

Listing 18.7 "molecule/default/tests/test_default.py": Molecule Tests with Testinfra Verifier

The name of a function that's intended for testing must also start with `test_`. As usual, you can start your tests now with this:

```
$ molecule verify
```

If you're not familiar with Testinfra, then if you're interested, you should take a look at the official documentation at *https://testinfra.readthedocs.io/en/latest*.

18.6 The Complete Test Cycle

The previous methods are very suitable for the iterative development of a role. If you need an automated process for the complete test cycle (including final cleanup), then you can perform it with this:

```
$ molecule test
```

If you occasionally need fresh test systems, then you can remove the current containers at any time by using this:

```
$ molecule destroy
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

18.7 Outlook and Conclusion

As in many sections in this book, I can't go into all the details of the tool we talked about here—particularly because Molecule represents more than just a tool; it might be an entirely new approach to your daily work. If you've decided that Molecule could enhance your development process, I recommend that you explore additional resources online.

A starting point is the official documentation at *https://ansible.readthedocs.io/projects/molecule/*, which might not be as thorough in certain areas. Also, you should take note of the *External Resources* at the bottom of the page; there are some very interesting talks and blogs linked there.

You also may be so intrigued by the topic that you steer your development process more towards *test-driven development*. In that type of software development, you *first* create the tests and *then* start implementing. As unconventional as it may seem initially, it does have the advantage of requiring you to clearly define your goals from the outset. Plus, you'll always know precisely when you can wrap up: when all tests are marked "OK."

So . . . happy testing!

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 19
Recipes, How-Tos, and Best Practices

You should think of this chapter as a collection of solutions to common practical challenges or as a source of inspiration and ideas for your own solutions. Each solution you'll find here isn't necessarily the best, most beautiful, or most efficient solution, but it's been tested and it works.

19.1 New Projects

In this section, we'll walk you through recommended settings as well as a template for new projects.

19.1.1 A Recommended "ansible.cfg"

The settings in Listing 19.1 have proven themselves, but of course, there's no configuration that is optimal for all environments. That's why I have set up the configurations for host key checking and SSH pipelining but left them commented out. Note that you should always review all settings in sample configuration files before using them in your own environment!

```
[defaults]
inventory = ./inventory
interpreter_python = auto_silent

#host_key_checking = False

stdout_callback = yaml

gathering = smart
fact_caching = jsonfile
fact_caching_connection = ~/.ansible/fact_cache
fact_caching_timeout = 86400

retry_files_enabled = yes
retry_files_save_path = ~/.ansible/retry-files

force_handlers = true
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
nocows = true

roles_path = ./roles

vault_id_match = true

ansible_managed = "Managed by ansible, don't make changes here!"

[ssh_connection]
#pipelining = true
```

Listing 19.1 "ansible.cfg": Recommended Configuration Settings

19.1.2 A Template for a New Project

If you prefer to use separate project directories when working with Ansible, you'll often find yourself needing to create new ones. This can be time-consuming, so it might be beneficial to automate the process.

For this purpose, I have created a small GitHub repository where you can find a shell script that you can use to create new projects.

You should clone as usual, like this:

```
$ mkdir -p ~/book; cd $_
$ git clone https://github.com/ansible-buch/init-ansible-project.git
```

You can then install the included script to your personal *bin/* folder:

```
$ cd init-ansible-project

$ mkdir -p ~/bin
$ install init-ansible-project.sh ~/bin
```

(If you didn't have a *bin/* folder before, you should log out and log back in so that your Linux will update the search path.)

Here's an example call:

```
$ cd ~/ansible/projects
$ init-ansible-project.sh test123

$ tree -a test123
test123
|-- ansible.cfg
|-- .envrc
|-- group_vars
|-- host_vars
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
|-- inventory
|-- playbooks
`-- roles
```

Take a close look at the resulting structure (especially the files) and adjust the script if necessary.

19.2 Administration

In this section, we'll walk through best practices for administration.

19.2.1 Building Simple Installers

Ansible is undoubtedly well suited for deploying software on systems, which might lead you to consider using it to create an installer—which is comparable to a *setup.exe* in the Windows world. Of course, this installer would be a playbook.

Naturally, the essential prerequisite is having the Ansible software, but that generally isn't an issue. Therefore, you could basically advise someone to install the *ansible* package and then run the playbook and go grab a coffee.

The requirements are therefore as follows:

- You want a playbook that is as easy to start as possible and that works on `localhost` without external dependencies (apart from Ansible)—without *ansible.cfg* and without inventory.
- The playbook should function effectively, even when such external conditions are present.
- Ultimately, root privileges are necessary for operation because the playbook is intended to make administrative changes to the local system.

You can see a suggestion for how you could solve this in Listing 19.2:

```
#!/usr/bin/env -S ansible-playbook -i localhost,
---
- hosts: localhost
  connection: local
  force_handlers: true

  vars:
    ansible_python_interpreter: auto_silent

  pre_tasks:
    - set_fact:
        eff_uid_number: "{{ lookup('pipe', 'id -u') }}"
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    - block:
      - debug: msg="Root privileges are required to run this command."
      - meta: end_play
      when: eff_uid_number|int != 0

  tasks:
    ### [Here we go. Do what needs to be done.] ###

    - name: Demo task (requires root privileges)
      command: test -r /etc/shadow
      changed_when: false
```

Listing 19.2 "installer.yml": Ansible-Based Installer

The key here is to make the playbook *executable* so that it can be run like a regular program:

```
$ chmod +x installer.yml
```

The shebang line provides the inventory information; otherwise, it would produce a warning message at startup. The `connection: local` setting prevents Ansible from working over SSH, which is its default method of operation. That would make little sense if you're already root and you perform something on `localhost`.

If you start the playbook as root (or prepend `sudo` or whatever is necessary to run a program with root privileges), then everything should work:

```
# ./installer.yml
[…]
TASK [Demo task (requires root privileges)] ********************************
ok: [localhost]
[…]
```

Here's what happens when an unprivileged user attempt is caught:

```
$ ./installer.yml
[…]
ok: [localhost] => {
    "msg": "Root privileges are required to run this command."
}
[…]
```

You could have used `meta: end_play` instead of `fail:`, but the output produced might confuse less experienced users.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

19.2.2 Determining the IP Address of a Target Host

Determining the IP address of a target host is a surprisingly persistent issue. The root of the problem lies in the fact that a host typically has multiple IP addresses (even when considering only traditional Internet Protocol version 4 [IPv4]), and Ansible cannot inherently determine which one is "the best." Let's explore the available options.

"ansible_default_ipv4"

This Ansible fact seems very promising at first. You should start by trying the following simple playbook in our lab environment:

```
---
- hosts: all

  tasks:
    - debug: var=ansible_default_ipv4.address
```

Listing 19.3 "default_ipv4.yml": Simple Determination of IP Address

Unfortunately, in our lab environment, you'll get addresses that you don't want:

```
ok: [rocky] => {
    "ansible_default_ipv4.address": "10.0.2.15"
}
ok: [debian] => {
    "ansible_default_ipv4.address": "10.0.2.15"
}
ok: [suse] => {
    "ansible_default_ipv4.address": "10.0.2.15"
}
ok: [ubuntu] => {
    "ansible_default_ipv4.address": "10.0.2.15"
}
```

The reason is that `ansible_default_ipv4` does *not* know (by magic or other means) the main IP of a server; rather, it is the IP address of the interface where the default route is directed. In rare cases, `ansible_default_ipv4` might even be undefined, but in our situation, these unusual results occur because the Vagrant provisioner has configured the systems in this manner.

My conclusion is that you should use this fact with caution!

Using the IP Address of a Specific Interface

Sometimes, you already know which interface your desired IP address belongs to (as in the following example on `eth1`). The key is to properly access the facts, as follows:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts: all

  vars:
    iface_name: eth1

  tasks:
    - debug: var=ansible_facts[iface_name]['ipv4']['address']
```

Listing 19.4 "interface_ipv4.yml": Determining IP Address of Specific Interface

The result is more in line with our expectations:

```
ok: [debian] => {
    "ansible_facts[iface_name]['ipv4']['address']": "192.168.150.10"
}
ok: [rocky] => {
    "ansible_facts[iface_name]['ipv4']['address']": "192.168.150.20"
}
ok: [suse] => {
    "ansible_facts[iface_name]['ipv4']['address']": "192.168.150.30"
}
ok: [ubuntu] => {
    "ansible_facts[iface_name]['ipv4']['address']": "192.168.150.40"
}
```

Using the IP Address of the Ansible Connection

When searching through all the host's details for the desired IP, you might encounter a third source: the `SSH_CONNECTION` environment variable. The content is as follows:

```
"SSH_CONNECTION": "192.168.150.129 55542 192.168.150.30 22"
```

Column 3 is the IP of the target host, but a quick test will show that this environment variable is not available on all lab hosts. Specifically, it's missing on those that use `sudo`; when you use privilege escalation, this variable is disregarded.

But we are determined to push it, so we simply add a configuration setting like the one in Listing 19.5, allowing `sudo` to keep this variable during privilege escalation:

```
---
- hosts: all

  tasks:
    - stat:
        path: /etc/sudoers
      register: sudoers
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
    - name: Add env_keep setting to /etc/sudoers
      lineinfile:
        path: /etc/sudoers
        line: "Defaults env_keep += SSH_CONNECTION"
        insertafter: "^Defaults"
      when: sudoers.stat.exists

    - name: Reload facts
      setup:

    - debug:
        msg: >-
          {{ ansible_facts['env']['SSH_CONNECTION'].split(' ')[2] }}
```

Listing 19.5 "connection_ipv4.yml": Determining IP Address via Environment Variable "SSH_CONNECTION": Variant with "SSH_CONNECTION"

And with that, it should work:

```
ok: [debian] => {
    "msg": "192.168.150.10"
}
ok: [rocky] => {
    "msg": "192.168.150.20"
}
ok: [suse] => {
    "msg": "192.168.150.30"
}
ok: [ubuntu] => {
    "msg": "192.168.150.40"
}
```

If you find this method interesting, you should make sure from the start that your target systems have this `sudo` configuration setting.

19.2.3 Managing "firewalld"

At least on Red Hat and SUSE systems, *firewalld* has now established itself as the standard firewall solution, and of course, there's also an Ansible module for interacting with this service. However, the way to resolve the issue of target hosts often having significantly different firewall setups is not entirely straightforward: on some target hosts, *firewalld* is active, on others, it is inactive or not present at all.

The playbook in Listing 19.6 shows an example of how to open certain ports with the `firewalld` module *if* the service is present and active:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts: all

  tasks:
    - name: Gather service facts
      service_facts:

    - name: Open HTTP(S) ports, if firewalld is active
      firewalld:
        service: "{{item}}"
        state: enabled
        permanent: yes
        immediate: yes
      with_items: [ http, https ]
      when: >
        ansible_facts['services']['firewalld.service'] is defined
          and
        ansible_facts['services']['firewalld.service']['state'] == "running"
```

Listing 19.6 "firewalld.yml": Opening Ports If "firewalld" Is Active

You can also solve this in a more classic way by simply checking the service status at the shell level, as Listing 19.7 shows:

```
---
- hosts: all

  tasks:
    - name: Check status of firewalld service
      shell: systemctl is-active firewalld.service || true
      register: firewalld_status_cmd
      changed_when: false

    - name: Open HTTP(S) ports, if firewalld is active
      firewalld:
        service: "{{item}}"
        state: enabled
        permanent: yes
        immediate: yes
      with_items: [ http, https ]
      when: firewalld_status_cmd.stdout == "active"
```

Listing 19.7 "firewalld-v2.yml": Opening Ports if "firewalld" Is Active

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The significantly shorter execution time makes the second option favorable, and the `service_facts` module seems to be somewhat sluggishly implemented. But ultimately, which option you use comes down to personal preference.

19.2.4 Applying Linux Software Updates

Ansible users often express a desire for the fully automated application of software updates from distribution packages. Such an implementation is basically not overly difficult, but if you need to add an automatic reboot, it can make the process a bit more complex.

The techniques differ slightly depending on the distribution, so I will now present three playbooks for our three distribution families.

Let's begin with Debian/Ubuntu hosts in Listing 19.8:

```
---
- hosts:
    - debian
    - ubuntu

  tasks:
    - name: Update package index
      apt:
        update_cache: yes
        cache_valid_time: 3600

    - name: Dist upgrade
      apt:
        upgrade: dist

    - name: Reboot required?
      stat:
        path: /var/run/reboot-required
      register: reboot_required

    - name: Reboot
      reboot:
      when: reboot_required.stat.exists
```

Listing 19.8 "update-debian.yml": Applying Linux Software Updates on Debian/Ubuntu

In the Red Hat distribution, it might look like this:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts:
    - rocky

  tasks:
    - name: Update all packages
      dnf:
        name: '*'
        state: latest

    - name: Reboot required?
      command: dnf needs-restarting -r
      changed_when: false
      failed_when: false
      register: reboot_required

    - name: Reboot
      reboot:
      when: reboot_required.rc != 0
```

Listing 19.9 "update-rocky.yml": Applying Linux Software Updates on Rocky

Finally, we can look at the SUSE environment:

```
---
- hosts:
    - suse

  tasks:
    - name: Update all packages
      zypper:
        name: '*'
        state: latest

    - name: Reboot required?
      command: zypper needs-rebooting
      changed_when: false
      failed_when: false
      register: reboot_required

    - name: Reboot
      reboot:
      when: reboot_required.rc != 0
```

Listing 19.10 "update-suse.yml": Applying Linux Software Updates on SUSE

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

19.2.5 Initial Distribution of SSH keys

As you may know, SSH keys are essential for conveniently and securely managing your Linux target systems with Ansible. But what if no keys are installed on many target systems initially? In this case, manually distributing keys, as described in Chapter 1, Section 1.8, might not be practical or desirable.

Certainly, there is no issue with automating key distribution using Ansible. The *authorized_key* module handles the primary task and performs at least the same function as the `ssh-copy-id` command on the Linux command line. Since the module requires the key as a text parameter, the main challenge is to use an appropriate lookup. Additionally, the following prerequisites must be met:

- The inventory must already be configured correctly.
- You must install the *sshpass* Linux package on the control host. See Chapter 1, Section 1.7 if you need to.
- The required Linux password must be the same on all addressed target hosts.

Here's a playbook that you can use as a starting point:

```
---
- hosts: all
  gather_facts: no

  vars:
    public_key_filename: ~/.ssh/id_rsa.pub

  tasks:
    - name: Install key from {{public_key_filename}}
      authorized_key:
        user: "{{ansible_user}}"
        key:  "{{lookup('file', public_key_filename)}}"
```

Listing 19.11 "install-ssh-key.yml": SSH Key Distribution via Ansible

The first call must of course use Linux password authentication (option `-k`):

```
$ ansible-playbook install-ssh-key.yml -k
```

Fancy a Lab Test?

If you want to test the key distribution in the lab first but all your target systems are already equipped with the corresponding SSH key, then you can add the `state: absent` parameter to the module call in the previous playbook and let it run through once. Afterwards, you should remove the parameter.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Finally, you should terminate all persistent connections to restore the original state:

```
$ ansible all -m meta -a reset_connection
```

19.2.6 Passing Passwords on the Command Line

Sometimes, you're dealing with an environment where no SSH keys are available. Let's take it a step further and not even assume an inventory.

To simulate this in the lab, you should set up a completely new user account, ensure that the *sshpass* package is installed, and finally switch identities:

```
$ sudo useradd -m -s /bin/bash user2
$ sudo apt install sshpass
$ sudo su - user2
```

In Chapter 1, Section 1.7, you learned how to be prompted for the SSH password—but here, I'm repeating only the commands and not the explanations:

```
Do this once:
$ mkdir -p ~/.ssh && chmod 700 ~/.ssh
$ ssh-keyscan -t rsa debian > ~/.ssh/known_hosts

And everything should now work:
$ ansible all -i debian, -u vagrant -m ping -k
$ ansible all -i debian, -u vagrant -a "test -r /etc/shadow" -k -b
```

There's nothing new so far, but the intriguing question is whether there's a solution that allows batch processing instead of requiring interactive password entry. The quick option would be to pass the `ansible_password` variable as an extra variable:

```
$ ansible all -i debian, -u vagrant -m ping -e ansible_password=vagrant
```

However, having plaintext passwords on the command line os never a good idea. A more secure approach you could take might look like this:

1. Put the password in an environment variable of your choice:

   ```
   $ export ANSIBLE_PASSWORD=vagrant
   ```

   You should make sure that it doesn't end up in your shell's history. For example, you can run a Bash with the `HISTCONTROL=ignoreboth` setting; this way, any line you start with a space will not be added to the history.

2. You now obtain the value for the extra variable via `env` lookup from this environment variable:

   ```
   $ ansible all -i debian, -u vagrant -m ping \
       -e ansible_password="{{lookup('env', 'ANSIBLE_PASSWORD')}}"
   ```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Typing all of this is obviously not enjoyable, but if you have an inventory, you can certainly specify it there:

```
[test]
debian

[test:vars]
ansible_user = vagrant
ansible_become = yes
ansible_password = {{lookup('env', 'ANSIBLE_PASSWORD')}}
```

Listing 19.12 "demo-inv"

And with that, it's more fun again:

```
$ ansible all -i demo-inv -m ping
$ ansible all -i demo-inv -a "test -r /etc/shadow"
```

19.2.7 Ansible via a Gateway or Jump Host

Sometimes, Ansible can't directly access target hosts due to network restrictions, and that means it needs a connection through a gateway or jump host instead. You can still achieve your goal by skillfully utilizing SSH capabilities, and from a purely SSH standpoint, this setup is detailed in Appendix C, Section C.7.7.

Let's assume that your entire Ansible project is located on a workstation and that there's a gateway host between this workstation and the target hosts, which you cannot bypass using SSH. To work transparently with Ansible, you should follow these steps:

1. Install your SSH public key for Ansible on the target hosts *and* the gateway host.
2. Make sure your target hosts are set up with the following parameters:

   ```
   ansible_ssh_common_args='-o ProxyCommand="ssh -q -W %h:%p USER@GATEWAY"'
   ```

   Here, you should replace `USER@GATEWAY` with the actual login credentials for the gateway system.

19.3 Jinja Magic

In this section, we'll showcase some different applications of Jinja.

19.3.1 Extending Maps or Lists during Runtime

If you ever encounter the challenge of needing to extend a map or a list during runtime, it's just a matter of knowing how. You can extend lists in Jinja simply with `+`, and you can extend maps with the `combine()` filter, as follows:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
---
- hosts: localhost

  vars:
    numbers: [11, 22, 33]
    animals: {Dog: woof, Cat: meow, Mouse: squeak}

  tasks:
    - set_fact:
        more_numbers: [44, 55]
        more_animals: {Donkey: hee-haw, Dog: growl}

    - name: Extend list
      set_fact:
        numbers: "{{ numbers + more_numbers }}"

    - name: Extend map
      set_fact:
        animals: "{{ animals | combine(more_animals) }}"

    - name: Print list
      debug: var=numbers

    - name: Print map
      debug: var=animals
```

Listing 19.13 "append.yml": Extending Lists or Maps

And this is what the result will look like:

```
$ ansible-playbook append.yml
[…]
TASK [Print list] ***********************************************************
ok: [localhost] => {
    "numbers": [
        11,
        22,
        33,
        44,
        55
    ]
}
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
TASK [Print amp] ************************************************************
ok: [localhost] => {
    "animals": {
        "Cat": "meow",
        "Dog": "growl",
        "Donkey": "hee-haw",
        "Mouse": "squeak"
    }
}
[…]
```

19.3.2 Modifying and Combining the Elements of a List

This is another job for Jinja. As long as the modification is textual, you can achieve a lot with regular expressions—though you may often need to perform a final *flattening* of the list into text when the result is intended for a configuration file. The following playbook shows both:

```
---
- hosts: localhost

  vars:
    list:
      - Dog
      - Cat
      - Mouse

  tasks:
    - debug:
        msg: >-
          {{ list | map('regex_replace', '^(.*) , '>>>\1<<<') | join('|') }}
```

Listing 19.14 "generate_list_text.yml": Converting Lists to Text

The parentheses in the first parameter of `regex_replace` captures the text matched by the pattern inside. In the replacement part, which is the second parameter, it is accessed with `\1` (making it a so-called backreference).

```
$ ansible-playbook generate_list_text.yml
[…]
ok: [localhost] => {
    "msg": ">>>Dog<<<|>>>Cat<<<|>>>Mouse<<<"
}
[…]
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

The true skill here lies not so much in Ansible itself but in the use of regular expressions. In the long run, delving deeply into this subject is always beneficial, so you should refer to Appendix D as a good introduction to it.

19.3.3 Searching in a List of Maps

Consider the following task: You have a list of objects (i.e., maps) and you are looking for a specific object (or even several). One solution is to use the Jinja filter `selectattr()`, which is parameterized with the key name, comparison operator, and desired value:

```
---
- hosts: localhost

  vars:
    animals:
      - name: Lassie
        type: Dog
        age:  7
      - name: Fido
        type: Dog
        age:  5
      - name: Kitty
        type: Cat
        age:  9


  tasks:
    - set_fact:
        matches: >
          {{ animals | selectattr("name", "equalto", query_name) | list }}
    - set_fact:
        animal: "{{ matches | first }}"
      when: matches

    - debug:
        msg: "{{animal.type}} {{animal.name}} is {{animal.age}} years old."
      when: animal is defined

    - debug:
        msg: "The animal could not be found."
      when: animal is not defined
```

Listing 19.15 "selectattr.yml": Searching in List of Maps

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Here are two search examples:

```
$ ansible-playbook selectattr.yml -e query_name=Lassie
[…]
ok: [localhost] => {
    "msg": "Dog Lassie is 7 years old."
}

$ ansible-playbook selectattr.yml -e query_name=Oscar
[…]
ok: [localhost] => {
    "msg": "The animal could not be found."
}
```

19.3.4 Filtering an Attribute from a List of Maps

This time, you want to select a specific attribute from a list of maps (for example, to process this attribute in a loop). The Jinja `map()` filter does exactly that:

```
---
- hosts: localhost

  vars:
    animals:
      - name: Lassie
        type: Dog
      - name: Fido
        type: Dog
      - name: Kitty
        type: Cat


  tasks:
    - debug:
        msg: >-
          {{ animals | map(attribute='name') | join(', ') }}
```

Listing 19.16 "map.yml": Creating List with Specific Attribute

The result looks like this:

```
ansible-playbook map.yml
[…]
ok: [localhost] => {
    "msg": "Lassie, Fido, Kitty"
}
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

19.3.5 Generating a Report from Target System Facts

Sometimes, you want to collect certain information in your target systems and record it in a file (usually on the control host). Listing 19.17 shows a simple example of how you could proceed.

The straightforward part is the initial play, where the root partition usage on all target hosts is identified as a demonstration. In the second play, you need to access these gathered facts using Jinja; the key elements are the implicit `groups` and `hostvars` variables (see also Chapter 6, Section 6.3.3).

```
---
- hosts: all

  tasks:
    - shell: df -h /
      register: df_cmd
      changed_when: false


- hosts: localhost

  tasks:
    - name: Write report
      copy:
        dest: /tmp/report.txt
        content: |
          {% for host in groups['all'] %}
          Result for '{{host}}':
          {{ hostvars[host]['df_cmd']['stdout'] }}

          {% endfor %}
```

Listing 19.17 "report.yml": Creating Report File

The result looks something like this:

```
$ ansible-playbook report.yml && cat /tmp/report.txt
[…]
Result for 'rocky':
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda2        62G  2.2G   60G   4% /

Result for 'suse':
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda3        64G  2.0G   58G   4% /
[…]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

19.3.6 Generating Passwords and Password Hashes

There are many ways to generate (random) passwords, and an (almost) all-in-one solution is to use the `password()` lookup plug-in.

The smartest feature of this plug-in is that it writes the generated plaintext password to a local file (in case you want to send the new password to your users via email later, for example). If this file already exists, the plug-in reads the password information from it, thus ensuring idempotency. Furthermore, you can determine the password length and complexity and also decide whether the generated output should be hashed or given in plaintext.

Consider the example from Listing 19.18 as a basis for your project. The generated passwords per user are located in the */tmp/credentials/* folder after the run. Of course, you should remove these files later:

```
---
- hosts: debian

  vars:
    users:
      - name: john
        fullname: John Doe
      - name: mary
        fullname: Mary Major
      - name: hans
        fullname: Hans Hansen

  tasks:
    - name: Create user
      user:
        name: "{{item.name}}"
        comment: "{{item.fullname}}"
        shell: /bin/bash
        password: >-
          {{ lookup('password',
                    '/tmp/credentials/pass_'
                    + item.name + '.txt'
                    + ' length=12 encrypt=sha512_crypt') }}
      with_items: "{{users}}"
```

Listing 19.18 "random-passwords.yml": Generating Random Password and Creating Linux Users with Random Passwords

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

This plug-in operates independently of its specific application. Whether you're creating Linux users or MySQL users or you need passwords for any other purpose, it's the surrounding module that defines how each password is utilized.

You can read more about this at *https://docs.ansible.com/ansible/latest/collections/ansible/builtin/password_lookup.html*.

If your concern is more about generating a hash value from an existing plaintext password, you should use the Jinja `password_hash()` filter. The following playbook exemplifies two variants of this:

```
---
- hosts: localhost

  vars:
    password: Supersecret

  tasks:
    - name: SHA512-hash (random salt)
      debug:
        msg: "{{ password | password_hash('sha512') }}"

    - name: SHA512-hash (idempotent)
      debug:
        msg: >-
          {{ password |
             password_hash('sha512', 65534   |
             random(seed=inventory_hostname) |
             string) }}
```

Listing 19.19 "password-hash.yml": Generating Password Hashes

19.4 Tasks and Control Flow

In this section, we'll walk through best practices for tasks and controlling execution flow.

19.4.1 Executing a Task Depending on a Previous Task

The key to this approach is registering the result of the previous task in a variable. That way, in subsequent tasks, you can make decisions based on the results of this task (e.g., executing only if there is a change):

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- hosts: localhost

  tasks:
    - name: Create directory /tmp/test123
      file:
        path: /tmp/test123
        state: directory
      register: file_task

    - debug:
        msg: "There was a change!"
      when: file_task.changed
```

Listing 19.20 "do_when.yml": Executing Task Only If There Is Change in Preceding Task

In essence, you created a handler, except that you don't have to wait until the end of the play for it to execute.

As you know, the `when` statement in Ansible requires a Boolean expression. You should therefore pay special attention if you have the text representation of a truth value, in which case, you must convert the text with the `bool()` filter:

```
---
- hosts: localhost

  tasks:
    - command: "echo 'No'"
      register: cmd_task

    - debug:
        msg: "It said 'Yes'!"
      when: cmd_task.stdout | bool
```

Listing 19.21 "do_when_bool.yml": Application of "bool" Filter

If you were to omit `| bool`, the `debug` task would be executed falsely because any non-empty text is evaluated as `True` in a Boolean context.

The `bool()` filter is pretty smart because it converts values like `"0"`, `"false"`, `"no"`, `"1"`, `"true"`, and `"yes"` (regardless of case) into appropriate Boolean values.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

19.4.2 Executing a Task if the Host Is in a Specific Group

Suppose you want to execute a task only if the current host is in a specific group (e.g., `some_group`). You can solve the problem by utilizing the implicit `group_names` variable (see Chapter 6, Section 6.3.3); it contains the list of all groups the host is a member of:

```
---
- hosts: all

  tasks:
    - debug: msg="I am a member of some_group"
      when: "'some_group' in group_names"
```

Listing 19.22 "task_for_group.yml": Executing Task If Host Is in Group

As usual, it's easy when you know how. However, applying the correct quoting of the `when` condition is not trivial.

19.4.3 Avoiding Redundant Module Parameters with "module_defaults"

Sometimes, when you're repeatedly calling the same module, you'll have redundant parameters—like the trio of `owner`, `group`, and `mode` in the following example:

```
---
- hosts: localhost

  tasks:
    - name: Create file1
      file:
        path: /tmp/file1
        state: touch
        owner: root
        group: root
        mode: 0755

    - name: Create file2
      file:
        path: /tmp/file2
        state: touch
        owner: root
        group: root
        mode: 0755
```

Listing 19.23 "module-without-defaults.yml": Redundant Parameters in Module Calls

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

You can avoid such redundancies with the `module_defaults` keyword, which works at the task or block level:

```
---
- hosts: localhost

  module_defaults:
    file:
      owner: root
      group: root
      mode: 0755

  tasks:
    - name: Create file1
      file:
        path: /tmp/file1
        state: touch

    - name: Create file2
      file:
        path: /tmp/file2
        state: touch
        mode: 0644
```

Listing 19.24 "module-with-defaults.yml": Less Redundancy with "module_defaults"

If you still want to pass parameters that deviate from the default, you can of course do so (see the `mode` parameter in the second task of this example).

You need to apply `module_defaults` at the block level, e.g., in roles. It might look like the following code snippet:

```
[...]

- name: Create some files
  module_defaults:
    file:
      owner: root
      group: root
      mode: 0755
  block:
    - name: Create file1
      file:
        path: /tmp/file1
        state: touch
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    - name: Create file2
      file:
        path: /tmp/file2
        state: touch

[...]
```

Listing 19.25 "tasks/main.yml": Example Excerpt of "module_defaults" at Block Level

19.4.4 Dynamically Set Play Hosts

So far, you've always specified the target hosts of a play statically:

```
- hosts: all

# [...]
```

Listing 19.26 "test.yml"

In addition to `all`, you can use any kind of pattern (see Chapter 3, Section 3.6). Nevertheless, it is still static information, and if you want to change the hosts, you have to change the playbook. Perhaps it is not a surprise that you can also work with Jinja here. Just choose a nice variable name:

```
- hosts: "{{ targets }}"
```

However, you must always set the variable when running the playbook; otherwise, you'll get the familiar `undefined variable` error message:

```
$ ansible-playbook test.yml
ERROR! The 'hosts' field has an invalid value that includes
an undefined variable. The was: 'targets' error is undefined.

$ ansible-playbook test.yml -e targets="debian,ubuntu"
```

If there's a meaningful default, you could of course specify it with a `default()` filter. Otherwise, using `default(omit)` is a good idea to help you avoid frightening error messages:

```
- hosts: "{{ targets | default(omit) }}"
```

If you're only bothered by the somewhat cumbersome parameterization with a non-standardized variable, you can take advantage of the Ansible limit mechanism, which you can control via the `--limit|-l` command line switch. The specified hosts are then available in the special `ansible_limit` variable:

```
- hosts: "{{ ansible_limit | default(omit) }}"
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

You can then make calls as follows:

```
$ ansible-playbook test.yml
PLAY ***************************************************************************
skipping: no hosts matched

$ ansible-playbook test.yml -l debian,ubuntu
PLAY [debian,ubuntu] ***********************************************************
[…]
```

19.4.5 Reading Configuration Files

Sometimes, you may want to access external configuration files to obtain information for the execution of the play. We assume that these configuration files are located on the control host, as the following solutions work exclusively with lookup plug-ins.

We consider three typical or common configuration formats, as explored in the following sections:

- YAML Ain't Markup Language (YAML)
- Extensible Markup Language (XML)
- Initialization (INI)

YAML

```
---
workshop:
  title: Configuration management with Ansible
  date:  "2024-03-19"
```

Listing 19.27 "config.yml": Example YAML Configuration File

The following playbook shows how you can read such files in YAML format. The first method is favored for its simplicity, while the second method is somewhat safer because you provide the namespace into which the configuration information is read:

```
---
- hosts: localhost

  tasks:
    - name: Read YAML config; method 1
      include_vars: "config.yml"

    - debug:
        msg: "Title: {{workshop.title}}"
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
    - name: Read YAML config; method 2
      set_fact:
        config: "{{ lookup('file', 'config.yml') | from_yaml }}"

    - debug:
        msg: "Title: {{config.workshop.title}}"
```

Listing 19.28 "read-yaml-file.yml": Reading YAML File

XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<workshop>
  <title>Configuration management with Ansible</title>
  <date>2024-03-19</date>
</workshop>
```

Listing 19.29 "config.xml": Example XML Configuration File

The `ansible.utils.from_xml` filter allows reading files in XML format. However, certain prerequisites must be met; see the comment in the following playbook:

```
---
- hosts: localhost

  tasks:
    # Requires:
    # - Python module xmltodict
    # - collection ansible.utils >= 2.0.2

    - set_fact:
        config: "{{ lookup('file', 'config.xml') | ansible.utils.from_xml }}"

    - debug:
        msg: "Title: {{config.workshop.title}}"
```

Listing 19.30 "read-xml-file.yml": Reading XML File

INI

```
[workshop]
title = Configuration management with Ansible
date  = 2024-03-19
```

Listing 19.31 "config.ini": Example INI Configuration File

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

You can read INI files in a convenient way with the `community.general.from_ini` filter, but you'll need a *very* new `community.general` collection. If you can't or don't want to meet this requirement, you can always resort to method 2, which has always been an option. (Though admittedly, it's a bit cumbersome if you need to read a large number of keys.):

```
---
- hosts: localhost
  gather_facts: no

  tasks:
    - name: Method 1; requires community.general Version >= 8.2.0
      block:
      - set_fact:
          config: >-
            {{ lookup('file', 'config.ini') | community.general.from_ini }}

      - debug:
          msg: "Title: {{config.workshop.title}}"

      ignore_errors: True

    - name: Method 2 (only targeted access is possible)
      set_fact:
        workshop:
          title: >-
            {{ lookup('ini', 'title', section='workshop',
               file='config.ini') }}
          date: >-
            {{ lookup('ini', 'date', section='workshop',
               file='config.ini') }}

    - debug:
        msg: "Title: {{workshop.title}}"
```

Listing 19.32 "read-ini-file.yml": Reading INI File

19.5 Miscellaneous

Now, we'll take a look at two other important skills: simulating functions and managing host-specific resources.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

19.5.1 Simulating Functions

If you have some programming experience, you may occasionally wonder how to implement the useful concept of a "function" in Ansible. Roles might seem a bit too heavyweight for you, and programming Ansible modules is certainly not everyone's cup of tea.

In such cases, I quite like working with includes, as the following example shows:

```
---
- hosts: localhost

  vars:
    # Set default name for the fact containing the result:
    result_fact: res

  tasks:
    - include_tasks: tasks/add.yml
      vars:
        x: 3
        y: 4

    - debug:
        msg: "add()-Result: {{res}}"

    - include_tasks: tasks/join.yml
      vars:
        list:
          - Dog
          - Cat
          - Mouse
        sep: "#"
        result_fact: joinresult

    - debug:
        msg: "join()-Result: {{joinresult}}"
```

Listing 19.33 "invoke-functions.yml": Playbook with Function Calls

The individual "functions" are then implemented as task files accordingly:

```
---
- set_fact:
    "{{ result_fact }}": "{{ x+y }}"
```

Listing 19.34 "tasks/add.yml": "add-function"

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
---
- set_fact:
    "{{ result_fact }}": "{{ list | join(sep) }}"
```

Listing 19.35 "tasks/join.yml": "join function"

You're not limited to simple Jinja—your functions can, of course, execute any Ansible tasks. In this specific example, you can achieve the desired effect:

```
$ ansible-playbook invoke-functions.yml
[…]
ok: [localhost] => {
    "msg": "add()-Result: 7"
}

[…]
ok: [localhost] => {
    "msg": "join()-Result: Dog#Cat#Mouse"
}
```

19.5.2 Managing Host-Specific Resources

Now, we come to a question that is not quite typical, but you may face it eventually. To set the stage, let's revisit our Apache role from Chapter 8, Section 8.4.

As you may remember, we used this role to set up an Apache web server on all target systems and equip it with a "pretty" start page. The new requirement is that some specific hosts need additional, individual HTML pages.

For example, the SUSE system should receive the following HTML page:

```
<h2>Info:</h2>
<p>SUSE Linux was first released in the mid-1990s.</p>
```

Listing 19.36 "info.html"

And the Rocky system might need to receive this HTML page:

```
<h2>Info:</h2>
<p>Rocky Linux is compatible with Red Hat Enterprise Linux.</p>
```

Listing 19.37 "info.html"

Now, an organizational question arises: Where should we store these files on the control host?

The *files/* or *templates/* folders of the role seem unsuitable because the role should remain independent of specific target systems. Just imagine that someone else is

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

developing and maintaining the role and that they know nothing about your specific target systems. They only want to incorporate the new feature into the role in coordination with you.

You know that Ansible offers an excellent solution for host-specific *variables*: the *host_vars/* folder. Essentially, we need a solution for host-specific files or resources, but unfortunately, best practices for this are scarce. Therefore, I'd like to propose a method that has consistently worked well for me.

You should set up a *resources/* hierarchy in the project's base folder, which is somewhat similar to `group_vars` and `host_vars`:

```
$ mkdir -p resources/group/{all,test_hosts}
$ mkdir -p resources/host/{rocky,suse}/htdocs
```

You can now place the two HTML files there. In the diagram, it would appear as follows:

```
resources
|-- group
|   |-- all
|   `-- test_hosts
`-- host
    |-- rocky
    |   `-- htdocs
    |       `-- info.html
    `-- suse
        `-- htdocs
            `-- info.html
```

And you can extend the Apache role with the following tasks:

```
- set_fact:
    resource_dir: "{{inventory_dir}}/resources/host/{{inventory_hostname}}"

- name: Upload info file if available
  block:
    - set_fact:
        filename: "{{resource_dir}}/htdocs/info.html"

    - stat:
        path: "{{filename}}"
      delegate_to: localhost
      register: local_content

    - copy:
        src: "{{filename}}"
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
        dest: "{{apache_document_root}}"
        mode: 0644
      when: local_content.stat.exists
```

Listing 19.38 "apache/tasks/main.yml": Additional Tasks for Apache Role

Unfortunately, there is no shorter (yet clean) way to copy a file only if it exists.

If you have a large number of files in the resource directories, you can proceed with a `with_fileglob` loop:

```
- set_fact:
    resource_dir: "{{inventory_dir}}/resources/host/{{inventory_hostname}}"

- name: Upload all files if available
  block:
    - set_fact:
        dirname: "{{resource_dir}}/htdocs"

    - copy:
        src: "{{item}}"
        dest: "{{apache_document_root}}"
        mode: 0644
      with_fileglob:
        - "{{dirname}}/*"
```

Listing 19.39 "apache/tasks/main.yml": Additional Tasks for Apache Role

We can omit the check with `stat` and the `when` condition on `copy` here: if there are no matches with `with_fileglob`, then it's not an error. There will be warning messages, but hopefully, they won't be too bothersome.

Unfortunately, this method is quite slow; uploading a large number of files can be quite tedious. If you're already (rightly) eyeing the `synchronize` module, then it would indeed be the far better choice because it is powered by `rsync` and you could even upload any directory hierarchies with it. Or, you could remove files from the target host if they are no longer present in the source folder.

But unfortunately, `synchronize` only works in conjunction with `become_user` when password-free `sudo` is the method for privilege escalation. This is another reason to make the `root` account directly accessible via an SSH -key on target systems for Ansible. (If it weren't for all those worrywarts . . . but that's a topic for another time.)

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Chapter 20
What Could Be Better, and What Is Still Missing?

Almost every software package has room for improvement, and Ansible is no different. In this section, I would like to address some missing features or shortcomings and, if possible, suggest a workaround for each.

20.1 Tracking Long-Running Tasks

Unfortunately, Ansible doesn't provide detailed progress updates for tasks while they're running. Tasks where this would be beneficial include downloading and copying files, as well as installing numerous distribution packages or packages with extensive dependencies.

The same thing applies to long-running commands, even if they would actually provide progress information when called outside of Ansible. All these outputs are only available upon the completion of the task, as you can illustrate with a playbook like the one in Listing 20.1:

```
---
- hosts: localhost

  tasks:
    - name: Long-running task
      shell: |
        for i in $(seq 10); do
            date | tee -a /tmp/date.log
            sleep 1
        done
      changed_when: false
      register: for_cmd

    - debug: var=for_cmd.stdout_lines
```

Listing 20.1 "long-running-task.yml": Long-Running Task

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

When called, it looks like this:

```
$ ansible-playbook long-running-task.yml
PLAY [localhost] ***********************************************************

TASK [Long-running task] ***************************************************
Wait ten seconds and then continue:
ok: [localhost]
[…]
```

Outside of Ansible, you have all of these options: while the playbook is running, you can observe processes on the target system, track file contents, or view directory listings. Using Ansible alone, you won't see a progress display; the internal architecture does not allow for it. The only way you could do it would involve quite a bit of tinkering with asynchronous tasks, but the effort is quite large, and you still wouldn't have a general solution. So, you should use the waiting time to grab a coffee and have a nice chat with your colleagues!

20.2 Finishing the Processing of a Role

So far, there is no *simple* way to stop processing a role at any given point.

Suppose you have a role with some tasks like these:

```
---
- debug: msg=Task1
- debug: msg=Task2
- debug: msg=Task3
- debug: msg=Task4
- debug: msg=Task5
```

Listing 20.2 "tasks/main.yml"

Furthermore, let's assume that you want to stop processing the role after the second task—either because you're still developing the role and Tasks 3 to 5 cannot work yet or because a certain condition arises that renders Tasks 3 to 5 unnecessary.

It would be ideal to have a counterpart to `meta: end_play` here, but unfortunately, it does not exist:

```
---
- debug: msg=Task1
- debug: msg=Task2

- meta: end_role  ### Unfortunately, this does not exist! ###
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
- debug: msg=Task3
- debug: msg=Task4
- debug: msg=Task5
```

Listing 20.3 "tasks/main.yml"

Of course, the `meta` directive could also be linked with a `when` condition, but—as mentioned—this is currently just wishful thinking. It has been a *feature idea* on GitHub since 2017.

What options do you still have to achieve our goal? You could work with a block:

```
---
- debug: msg=Task1
- debug: msg=Task2

- when: false
  block:
  - debug: msg=Task3
  - debug: msg=Task4
  - debug: msg=Task5
```

Listing 20.4 "tasks/main.yml"

Instead of `false`, you could evaluate any other condition. You might find it inconvenient to change the indentation of the tasks, but otherwise, this works great.

If this is a bit too confusing, you might resort to a conditional include:

```
---
- debug: msg=Task1
- debug: msg=Task2

- include_tasks: tasks_3-5.yml
  when: some_condition
```

Listing 20.5 "tasks/main.yml"

And there, quite simply, you do this:

```
---
- debug: msg=Task3
- debug: msg=Task4
- debug: msg=Task5
```

Listing 20.6 "tasks/tasks_3-5.yml"

This is certainly OK as a permanent solution if you don't mind carrying a second file.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

20.3 Loops over Blocks

Unfortunately, you cannot create loops over blocks in Ansible. In Listing 20.7, you'll see a playbook that is not executable in Ansible at the time of writing this book:

```
---
### Warning: This playbook is not executable! ###
- hosts: localhost

  tasks:
    - block:
      - debug: msg="Hello {{item}}"
      - debug: msg="Hello again {{item}}"
      with_items:
        - Dog
        - Cat
        - Mouse
```

Listing 20.7 "playbook-block-loop.yml": Unfortunately Not Functional!

```
$ ansible-playbook playbook-block-loop.yml
ERROR! 'with_items' is not a valid attribute for a Block
[…]
```

Here, too, the workaround is to outsource the corresponding tasks into a second file, as in Listing 20.8:

```
---
- hosts: localhost

  tasks:
    - include_tasks: sometasks.yml
      with_items:
        - Dog
        - Cat
        - Mouse
```

Listing 20.8 "playbook-include-loop.yml": Looping Over Outsourced Tasks

And there, you simply do this:

```
---
- debug: msg="Hello {{item}}"
- debug: msg="Hello again {{item}}"
```

Listing 20.9 "sometasks.yml": Outsourced Tasks to Include

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The evident drawback of this method is that it requires the involvement of a second file. Also, it's important to recognize that this solution isn't flawless either, for the following reasons:

Important

Looping over includes only work with `with_*` or `loop`.

Unfortunately, you cannot combine an `until` loop with an include!

20.4 Locking with Concurrent Playbook Calls

Are you running the same playbook twice at the same time? That usually works, but not always. For example, when you're attempting to modify the same file simultaneously, you can expect problems (at least theoretically).

Ansible doesn't provide a built-in locking mechanism, so when you need to synchronize, you need to implement the mechanism yourself. A solution could be something like this:

```
---
- name: A lock test
  hosts: localhost

  vars:
    lockfile_path: >
      /tmp/ansible-playbook-{{ansible_play_name | hash('md5')}}.lock

  pre_tasks:
    - stat:
        path: "{{lockfile_path}}"
      register: lockfile
    - fail:
        msg: "This playbook is probably already running."
      when: lockfile.stat.exists and not force_ignore_lock|default(false)
    - name: Create lockfile
      file:
        path: "{{lockfile_path}}"
        state: touch
    - debug:
        msg: "Lockfile created: {{lockfile_path}}"

  post_tasks:
    - name: Remove lockfile
      file:
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
        path: "{{lockfile_path}}"
        state: absent

  tasks:
    - pause:
        seconds: 10
```

Listing 20.10 "locktest.yml": Example of Locking at Playbook Level

The actual task of the playbook is simulated in the `tasks` section by a 10-second pause, and you should test it with two windows or two parallel logins. The invocation options are as follows:

```
$ ansible-playbook locktest.yml

Simply bypass any existing lock:
$ ansible-playbook locktest.yml -e force_ignore_lock=1
```

20.5 Conclusion

Even though you've encountered a couple of challenges in this chapter, it's a classic example of "complaining on a high level." Ansible has matured into a very well-functioning, well-thought-out, and powerful piece of software after well over a decade of development time, and using it is generally a lot of fun.

We have every reason to be satisfied and look forward to the developments the coming years will bring!

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Appendices

A Project-Specific Environment Variables with “direnv” 429

B The “pass” Password Manager 433

C Secure Shell 437

D Regular Expressions 455

E “vim” and “nano” Tips and Tricks 463

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Appendix A
Project-Specific Environment Variables with "direnv"

The *direnv* shell extension (*https://github.com/direnv/direnv*) is very useful because it allows you to set or modify environment variables depending on the current working directory—and it does so fully automatically (i.e., simply by changing into a suitably configured directory). When you leave such a directory area, the changes you make will be automatically removed.

A.1 Installation and Shell Integration

The *direnv* extension is packaged in Debian, Ubuntu, and openSUSE distributions, and you can easily install it via the package of the same name.

Regrettably, neither Rocky nor CentOS offers a package at the moment (not even in EPEL). The only option is to install it directly from the project's website, like this:

```
# VERSION="2.34.0"

# curl -sSL https://github.com/direnv/direnv/releases/download/\
v$VERSION/direnv.linux-amd64 \
-o /usr/local/bin/direnv

# chmod 755 /usr/local/bin/direnv
```

The *direnv* extension works with some popular shells, but we'll focus on Bash. It works its magic by being integrated via the `PROMPT_COMMAND` variable, which is a Bash feature that allows you to automatically call any command every time before a new prompt appears.

You can simply add a special *eval* line as the last action in your `~/.bashrc`:

```
# ...

eval "$(direnv hook bash)"
```

Listing A.1 "~/.bashrc": Activation via ".bashrc"

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

For instance, you can achieve this with the following command if you prefer not to use an editor:

```
$ echo 'eval "$(direnv hook bash)"' >> ~/.bashrc
```

If you want to activate this new setting in your Bash now without relogging, you can do so as usual with a `source` call:

```
$ source ~/.bashrc
```

A.2 Usage

You use *direnv* by placing an *.envrc* file in a directory, which contains variable definitions in the usual syntax. Try it:

```
$ mkdir testproject && cd $_
```

For example, you can place this file there:

```
export ANIMAL=Elephant
export NUMBER=42
```

Listing A.2 “.envrc”: Example “direnv” File

Whenever you’re in this directory (or in a subdirectory of it), these variables will be added to your working shell. After each time you change or create an *.envrc* file, *direnv* will cautiously ask you this:

```
direnv: error /home/vagrant/testproject/.envrc is blocked.
Run `direnv allow` to approve its content

$ direnv allow
direnv: loading ~/testproject/.envrc
direnv: export +ANIMAL +NUMBER

$ echo $ANIMAL $NUMBER
Elephant 42
```

When you leave, the variables will be removed:

```
$ cd ..
direnv: unloading
```

Modifying the `PATH` variable is often useful to make project-specific programs accessible:

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
PATH=~/programs/kubectl/1.21.5:$PATH

# or shorter with direnv special function:
PATH_add ~/programs/kubectl/1.21.5
```

Listing A.3 ".envrc": Example "direnv" File

All special functions are well documented at *https://github.com/direnv/direnv/blob/master/man/direnv-stdlib.1.md*. There, you'll also find the `expand_path` function, which converts a relative path into an absolute path:

```
export ANSIBLE_CONFIG=$(expand_path ansible.cfg)
```

Listing A.4 ".envrc": Example "direnv" File

This then gets the absolute path of the file as a value, for example:

```
$ echo $ANSIBLE_CONFIG
/home/vagrant/testprojekt/ansible.cfg
```

A.3 Aliases and Functions

Due to how *direnv* operates internally, you can't directly manage aliases or functions with it. However, with a slight workaround, you can achieve a similar result.

Suppose you want to create a project-specific `editor` alias that calls `vim` in one project and `nano` in another project.

In your *~/.bashrc*, you can first create a "global" function like this:

```
function editor () { ${PROJECT_EDITOR-vi}; }
```

Listing A.5 "~/.bashrc": Example ".bashrc" Excerpt

(Don't forget to activate with `source` or log in again.)

In the project-specific *.envrc* files, you can set the desired value as needed:

```
export PROJECT_EDITOR=nano
```

Listing A.6 ".envrc": Example "direnv" File

And now, when you call `editor`, the set editor should open!

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

A.4 Command Execution Upon Entry

Starting specific commands automatically upon entering a *direnv* environment is not fundamentally problematic. Ultimately, you start *.envrc* as a shell script, so all commands therein will be executed:

```
export ANIMAL=Elephant

echo "Hello $ANIMAL in the direnv environment!"
echo "Now we can get started."
```

Listing A.7 “.envrc”: Example “direnv” file

You should just keep two things in mind:

- Unfortunately, there’s no simple mechanism to run certain commands when you leave the environment (for example, to undo something).
- If you work with multiple shells and enter the environment each time, your commands will also be executed multiple times. If that is a problem, you’ll need to handle it manually.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Appendix B
The "pass" Password Manager

The *pass* password manager (*https://www.passwordstore.org/*) is a simple and secure command-line password manager that stores passwords and other sensitive data in an encrypted directory structure. The encryption of the data is done using GNU Privacy Guard (GPG).

B.1 Installation and Preparations

If you want to use *pass* for the first time, you must first create a GPG key pair. However, you'll often encounter a problem that's very typical for (minimal) virtual machines: the system can't gather *entropy* (i.e., truly random randomness) fast enough for `gpg --gen-key` to complete its task.

To largely avoid this problem, you should first install and activate the *haveged* software:

```
In Debian/Ubuntu, you do it this way:
# apt install haveged

In Rocky, you do it this way:
# dnf install haveged
# systemctl enable --now haveged

And in openSUSE, you do it this way:
# zypper install haveged
# systemctl enable --now haveged
```

You can also install the actual *pass* software quickly, although in Rocky 9, you may need to activate the EPEL first:

```
In Debian/Ubuntu, you do it this way:
# apt install --no-install-recommends pass

In Rocky, you do it this way:
# dnf install epel-release
# dnf install pass
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
And in openSUSE, you do it this way:
# zypper install password-store
```

Now, you should assume the identity of the user with whom you want to use password encryption. You should be able to generate a GPG key pair as follows:

```
$ gpg --gen-key
[…]

GnuPG needs to construct a user ID to identify your key.

Real name: First User
Email address: user1@example.org
[…]

+----------------------------------------------------+
| Please enter the passphrase to protect your new key |
|                                                    |
| Passphrase:                                        |
+----------------------------------------------------+
[…]
```

Warning

The passphrase must be at least eight characters long and contain at least one special character or digit. The prompt that asks whether you want to use an unsecure passphrase doesn’t work in some terminal environments, so without a strong password, you won’t be able to proceed!

If everything goes smoothly, you should be able to view your list of keys by using `gpg --list-keys`.

B.2 Usage

Now that all prerequisites are met, you can set up your password store:

```
$ pass init "First User"
```

Important: The string must uniquely match the appropriate GPG key!

By the way, Git experts can also use `pass git init ...` and place the password store under version control.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

You can now create password entries with any structure in the store:

```
$ pass insert shops/shop1
Enter password for shops/shop1:
Retype password for shops/shop1:

$ pass insert banks/bank1
[…]
$ pass insert ansible/vault/dev
[…]
$ pass insert ansible/vault/prod
[…]
```

You can then retrieve entries as follows:

```
$ pass shops/shop1
```

The GPG agent saves you from repeatedly entering your GPG passphrase.

With `pass edit <PATH>`, you can also enter multiline secrets; an editor will start.

If you just want to take a look at the current structure or hierarchy, you can just call `pass` or `pass ls`:

```
$ pass
Password Store
|-- ansible
|   `-- vault
|       |-- dev
|       `-- prod
|-- banks
|   `-- bank1
`-- shops
`-- shop1
```

These are by no means all the features of `pass`, but for our use case, they should suffice. You can find many more examples online; the included *pass(1)* man page is somewhat sparse but very good.

B.3 Configuring GNU Privacy Guard Agent Cache Times

The GPG agent is quite "forgetful" by default: if you don't use your key for 10 minutes (600 seconds), it will ask for the passphrase again—and after a maximum of 2 hours (7,200 seconds), it will do so in any case.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

You can configure these time periods in the *~/.gnupg/gpg-agent.conf* file, for example, to one hour or one day:

```
default-cache-ttl  3600
max-cache-ttl     86400
```

Listing B.1 “~/.gnupg/gpg-agent.conf”: Configuration of GPG Agent’s Cache Times

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Appendix C
Secure Shell

Secure Shell (SSH) has long been the de facto standard in the Linux world for performing typical tasks such as the following:

- Logging in to remote systems
- Executing commands on remote systems
- Copying files over a network

Although there were a plethora of commands or protocols for these purposes in earlier times (RSH, TELNET, FTP, etc.), they are now considered highly unsecure and vulnerable because both the passwords and all subsequent network traffic are typically transmitted unencrypted. On the other hand, the SSH suite offers a robust, reliable, and secure alternative that is also quite easy to use.

C.1 Server-Side Requirements for Using Secure Shell

You must make sure the SSH service and machine are up and running before you can connect to a Linux machine via SSH. This should be the case for most Linux installations, and you can verify this on the target machine with the following:

```
$ ps -ef | grep sshd
root      3078     1  0 Oct20 ?        00:00:00 /usr/sbin/sshd
```

or

```
$ ss -tnl | grep -w 22
LISTEN 0      128          0.0.0.0:22         0.0.0.0:*
LISTEN 0      128             [::]:22            [::]:*
```

The second check assumes that the service is running with the default configuration and is listening on TCP port 22. At this point, only firewalls on or in front of the machine could thwart your plans—but that's a topic for another time.

If you ever need to reinstall SSH on a Linux machine, please use the following packages:

- On Debian, Ubuntu, or Red Hat/Rocky, use *openssh-server*.
- On SUSE, use *openssh*.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

C.2 Secure Shell Client Programs

In the rest of this appendix, we'll focus almost exclusively on the "user side" of SSH. You'll see that there are a whole lot of different client tools, some of which depend on the operating system platform (Linux or Windows). First, let's explore the most essential command-line clients that are available on Linux and modern Windows systems:

- `ssh`
  You use this to execute remote commands (and also to log in remotely).
- `scp`
  You use this to copy files over the network.
- `sftp`
  You also use this to copy files over the network, and it's modeled after an FTP client.

C.2.1 "ssh": Remote Shell

```
ssh [-l <USER>] <HOST> [<COMMAND>]
ssh [<USER>@]<HOST> [<COMMAND>]
```

You can use `ssh` to execute commands on remote computers. If you don't specify a command, a login shell will be started.

Examples of remote command execution are as follows:

```
Output on a local terminal:
$ ssh user1@somehost.example.org  ls -l

Output redirection in which a file is created locally:
$ ssh user1@somehost.example.org  ls -l >ls.out

Remote creation of a file:
$ ssh user1@somehost.example.org "ls >ls.out"

Piping over the network:
$ echo hello | ssh user1@somehost.example.org "cat >hello.txt"
```

C.2.2 "scp": Remote Copy

```
scp [-r] <SOURCE...> <TARGET>
```

Both `<SOURCE>` and `<TARGET>` have the following format for a remote site:

```
[<USER>@]HOST:[PATH]
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

- If you omit the <USER>@ part, the local account will also be used for the remote system.
- If you omit the <PATH> part, the home directory of the remote account will be assumed.
- With the -r option, you can copy entire directory trees recursively.

Here are some examples:

```
Copying the 'file' file to the 'user1' user on somehost.example.org to /tmp:
$ scp file user1@somehost.example.org:/tmp
And copying it back again:
$ scp user1@somehost.example.org:/tmp/file .
```

C.2.3 "sftp": Secure File Transfer Protocol

```
sftp [<USER>@]<HOST>
```

The sftp command offers functionality that's comparable to a traditional file transfer protocol (FTP) client. It can be quite useful if you don't know the directory structure on the remote side—then, you can navigate with cd and ls and download or upload files with get and put. With the help command, you can see all available commands.

C.3 Public Key Authentication

By default, you must enter a password for all SSH commands to provide authentication. However, this isn't always feasible—either due to the inconvenience or for compelling reasons such as automation. Also, from a security perspective, passwords are not the first choice.

SSH offers a quite convenient and secure alternative here: *public key authentication*. Its technical background is *asymmetric encryption methods*: with a shared key that's split into a public part and a private part, you can encrypt messages with one part of the key so that they can only be decrypted with the corresponding other part.

You can employ this technique in SSH to verify people's identity: anyone who holds the private key that corresponds to a specific a public key will be authorized to access the system.

Figure C.1 illustrates this concept in simplified form.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

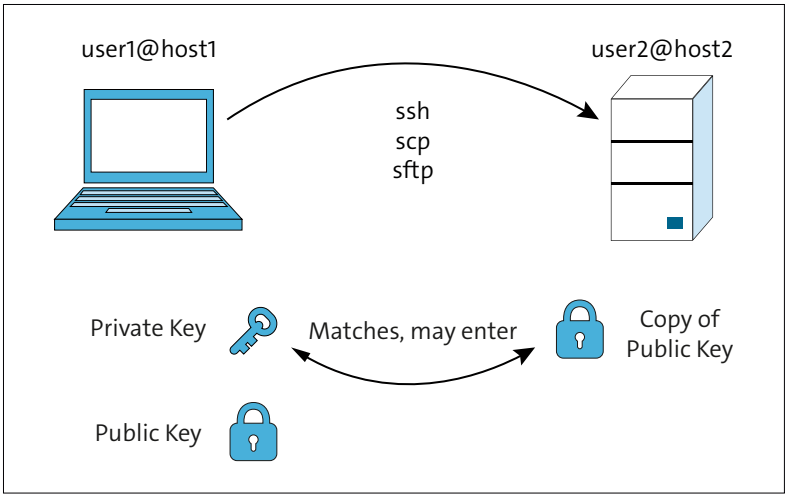


Figure C.1 SSH Public Key Authentication

C.3.1 Generating a Secure Shell Key Pair

Let's start by generating a key pair. You should add the `-b 4096` option to the `ssh-keygen` command if you plan to use this key for *many* years.

Important

The program will ask you for a *passphrase* that protects your private key—if someone steals it, they can only "decrypt" it through brute-force attacks. The longer and more complex the passphrase is, the longer that process will take, thereby giving you valuable time.

You should use an empty passphrase only in test environments or for automated tasks like cron jobs, and only if you fully understand the risks involved!

```
$ ssh-keygen
Generating public/private RSA key pair.
Enter file in which to save the key (/home/user1/.ssh/id_rsa): Enter
Created directory '/home/user1/.ssh'.
Enter passphrase (empty for no passphrase): __________
Enter same passphrase again: __________
Your identification has been saved in /home/user1/.ssh/id_rsa.
Your public key has been saved in /home/user1/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:ElUrrbvs51RUiWwECIeAXvAZSoYUokDxTuAhOdED+Sk user1@host
The key's randomart image is:
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

```
+---[RSA 2048]----+
|B@OBo..oooo+.... |
|BoBo+o.o.. .+..  |
|.o.=oo. . o..    |
|  E o  . o .     |
|   o  . S  .     |
|       . . .     |
|        . .      |
|       . o.      |
|       .+o.      |
+----[SHA256]-----+
```

If you haven't changed the file paths, the following two files will now be created:

- *~/.ssh/id_rsa.pub* (the public key)
- *~/.ssh/id_rsa* (the private key)

You can change the passphrase with the `ssh-keygen -p` command:

```
$ ssh-keygen -p
Enter file in which the key is (/home/user1/.ssh/id_rsa):
Enter old passphrase:
Key has comment 'user1@host'
Enter new passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved with the new passphrase.
```

C.3.2 Uploading the Public Key to the Target System

Now, you must upload the public key to a very specific location on the target system.

Uploading the Public Key

You must append the public key (i.e., the content of *id_rsa.pub*) to the *~/.ssh/authorized_keys* file on the *target account*.

There are at least three ways to accomplish this:

- **Manually**
  You log in to the target account, create the *~/.ssh* folder if necessary, and then start an editor like `vi` or `nano` with the *~/.ssh/authorized_keys* file to copy the key into it, using the mouse.
- **With the `ssh-copy-id` program (only on Linux)**
  This utility performs all necessary steps with one simple command:

  ```
  $ ssh-copy-id <USER@HOST>
  ```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

Warning

Please be cautious when agent forwarding is enabled (Section C.7.6)! In this case, `ssh-copy-id` will by default install the forward key—instead of the local public key, as you might have intended.

If you dig deeper into the topic, you'll find that this behavior makes a lot of sense. At first glance, however, the results appear utterly baffling.

- **With "simple" commands, without `ssh-copy-id`**

```
$ cd .ssh
$ ssh <USER@HOST> install -d .ssh -m 700
$ cat id_rsa.pub | ssh <USER@HOST> "cat >>.ssh/authorized_keys"
```

(On Windows, instead of the initial `cat`, you should use the equivalent `type`!)

Regardless of the method you've used, any subsequent SSH actions you take in this target account should now prompt you for the *key's passphrase* instead of the password. If your key doesn't have a passphrase, then no interaction will be required.

If this doesn't work, keep in mind that SSH refuses public key authentication if the home directory of the target account is writable for anyone other than the owner. (The default setting is `StrictModes yes`.) The permissions of the *.ssh* folder should also be as restrictive as possible, so please check this.

Also, pay attention to access control lists (ACLs)—depending on the SSH server version, you should also check these!

C.3.3 Public Key Distribution with "ssh-copy-id"

By default, the familiar `ssh-copy-id` tool is designed to distribute only those keys for which the private part is also accessible. However, with the appropriate parameterization, you can distribute any public key. Assuming you have a public key in the *key.pub* file, here's how to upload it to another Linux system:

```
ssh-copy-id -f -i key.pub <USER@OTHERHOST>
```

Note

Using the `.pub` file extension is very important here. If you specify a filename that doesn't have this extension, `ssh-copy-id` would internally add the `.pub` extension before searching for the file—which would probably lead to a dead end.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The `-n` to `ssh-copy-id` option is also very useful. If you specify it in addition to the required parameters, you'll initiate a dry run (i.e., a simulation). The output will show you what *would happen* in normal operating mode, so you can verify the correctness before actually making changes to your target systems.

C.4 SSH Agents (Linux Client)

Instead of constantly entering passphrases for private keys, you can use a so-called *SSH agent*. This is a background process that keeps a selection of private keys in main memory and automatically responds to corresponding requests.

Some Linux desktops automatically launch an SSH agent when you log in. To check if your setup offers this, you can use this:

```
$ echo $SSH_AGENT_PID
```

If you see a process identifier (PID), you can direct the agent with `ssh-add` to manage your key.

If not, there are two ways to invoke and use an SSH agent from a shell:

- The single-shell method
- The subshell method

Important

Do **not** start the agent in what you might think is the obvious way, just by using this command:

```
ssh-agent
```

It will run, but SSH clients will not be able to easily connect with it, and you won't be able to easily terminate it (except via `ps`/`kill`)!

Which method you should choose largely depends on your personal preference.

C.4.1 Start with the Single-Shell Method

The advantage of the single-shell method is that you stay in the same shell, which allows you to use old commands from the current history. Despite the somewhat cumbersome invocation with `eval`, I also recommend that you do this:

```
$ eval $(ssh-agent)
Agent pid 32001

Passing the key:
$ ssh-add
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
Enter passphrase for key '/home/account/.ssh/id_rsa': ________
Identity added: /home/user1/.ssh/id_rsa (/home/user1/.ssh/id_rsa)

[... working with ssh, scp, sftp ...]

Terminating the agent:
ssh-agent -k
```

C.4.2 Starting with the Subshell Method

With the subshell method, you implicitly get a new working shell. The procedure is as follows:

```
$ ssh-agent bash

$ ssh-add
Enter passphrase for key '/home/account/.ssh/id_rsa': ________
Identity added: /home/user1/.ssh/id_rsa (/home/user1/.ssh/id_rsa)

[Then, you work with ssh, slogin, scp, sftp, etc.]

Then, you terminate the agent:
exit
```

C.5 PuTTY, PuTTYgen, and Pageant

The free *PuTTY* software (at *https://www.putty.org*) is a widely used and reliable alternative to the built-in command line tools in Windows, which have only been available since Windows 10.

For installation, it is easiest to use the Windows installer. Only in exceptional cases (e.g., no local administrator rights) would you download individual components like `putty.exe` from the website.

C.5.1 PuTTY

The main application of the *PuTTY* software is logging into remote systems. For that, you use the tool with the same name: *PuTTY*.

I highly recommend that you to use *sessions* here to handle specific configurations of various target systems. Some useful session settings include the following:

- **Window • Appearance**
  You use this to set font and size.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

- **Window • Translation**
  You use this to select the encoding of the character set, which is typically UTF-8 nowadays.
- **Window • Translation, Enable VT100 line drawing even in UTF-8 mode**
  You use this to fix display errors in applications like `yast`, `mc`, and `ncdu` (in PuTTY version 0.71 and later).
- **Window • Colors**
  You use this to color settings. Tip: **Use System Colors** results in black on white!
- **Connection • Data, Auto-login username**
  This saves you from repeatedly entering the username, which is especially useful with public key authentication!

C.5.2 PuTTYgen

Of course, you can manage keys with *PuTTY*, too. To generate or edit keys, you need the *PuTTYgen* tool:

- You can create a new key pair with **Generate**.
  You can then optionally set a passphrase and permanently save the key with **Save private key**. (It includes the public part. It is unlikely that you'll ever need to use the **Save public key** button because the generated format would have to be converted to function with OpenSSH anyway.)

  Then, you copy the public part of the key, which you need to upload to the target account. You do this with the mouse in the field under **Public key for pasting into OpenSSH authorized_keys File**.
- You can edit existing PuTTY keys by opening the corresponding *.ppk* file with **Load**. Now, you could, for example, change the passphrase or the comment.
- You can open a key in OpenSSH format (which you have generated with the `ssh-keygen` command) with **Conversions • Import key** to save it afterwards in PuTTY format, for example.

C.5.3 Pageant

The SSH agent works more smoothly in PuTTY than in the command line environment. In PuTTY, the agent is available as the *Pageant* tool:

- To start the agent and feed it with a key, you can just double-click on the corresponding key file (if the *.ppk* file extension is properly associated).
- You can manage a running agent via the icons at the bottom right of the taskbar.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

C.5.4 Additional Tools

The PuTTY suite also includes the `pscp.exe` and `psftp.exe` command-line tools, which are PuTTY counterparts for file transfer.

And last but not least, there is `plink.exe`, which you can use for noninteractive command execution.

C.6 Windows Secure Copy

Windows Secure Copy (WinSCP) is a free SFTP client for Windows at *http://winscp.net/*. It's commonly employed for secure and convenient file transfers between Windows and Linux systems.

WinSCP is "PuTTY-compatible," meaning it can import PuTTY sessions and, most importantly, work with the PuTTY agent!

One of the many other conveniences of WinSCP is its ability to transparently access Linux files with Windows editors. The included internal editor is quite rudimentary, but you can also link any external editors like Notepad++.

To do this, you select **Settings • Preferences** and then select the **Editors** subsection.

Then, you click on **Add** and select the external editor from the system.

You should then move this to the top of the list of available editors so that you can use it by default when you do something like double-clicking a file.

WinSCP can only access the permissions that the authenticated user has on the target Linux system. In cases where this user serves merely as an intermediary to the root user, there may be a need to operate transparently as root in WinSCP. Since direct root login is typically disabled, certain workarounds will be necessary.

In any case, the prerequisite is that the nonprivileged user can achieve elevated rights with *password-free* `sudo`!

If that is the case, you can go to the corresponding connection target under **Advanced...**, find the **Environment • SFTP** section, and then enter the following specific to the distribution at **SFTP server**:

- **Debian**

  ```
  sudo /usr/lib/openssh/sftp-server
  ```

- **Rocky Linux 9**

  ```
  sudo /usr/libexec/openssh/sftp-server
  ```

- **openSUSE Leap 15**

  ```
  sudo /usr/lib/ssh/sftp-server
  ```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

C.7 Advanced Configuration and Usage

C.7.1 Managing Access

In the */etc/ssh/sshd_config* server configuration file, you can adjust settings such as these:

- **Prohibiting or allowing root login/not with UNIX password**

  `PermitRootLogin no|yes|without-password`

- **Allowing or prohibiting UNIX password authentication in general**

  `PasswordAuthentication yes|no`

- **Granting access only to specific users or groups**

  ```
  AllowUsers user1 user2 ...
  AllowGroups group1 group2 ...
  ```

  The evaluation order is `DenyUsers`, `AllowUsers`, `DenyGroups`, `AllowGroups`.

C.7.2 Conditional Configuration with "Match"

With the `Match` keyword, you can implement conditional configuration sections in the server configuration. Such a `Match` block extends to the next `Match` statement or to the end of the file.

Here's an example in which password authentication is generally disabled but still permitted for specific users:

```
[...]
PasswordAuthentication no
[...]

Match User john,harry
    PasswordAuthentication yes
```

Listing C.1 Prohibiting Password Authentication with Few Exceptions

The indentation in `Match` blocks is purely cosmetic; it is unnecessary for the parser and is ignored. In the comma-separated name lists, *no spaces* are allowed; some caution is advised here.

Besides matching on `User`, you can of course also match on `Group`, and there are other criteria such as `Host` or `Address`. Please refer to the *sshd_config(5)* man page if you need to; it also describes which keywords can be used in a `Match` block.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

C.7.3 Client Preferences

You can set client preferences per user in the *~/.ssh/config* file or as a global default in the */etc/ssh/ssh_config* file:

```
Host server1
     HostName server1.example.org
     User root
     Port 2222

Host server2
     [...]
```

Listing C.2 "~/.ssh/config": Setting Host-Specific Preferences

You can configure settings that are applicable to all hosts in a default section:

```
Host *
     [...]
```

Listing C.3 "~/.ssh/config": Setting Defaults for All Hosts

See also *ssh_config(5)* for a complete description of the options.

C.7.4 Verifying Fingerprints

You probably know this prompt:

```
ssh user@192.168.150.10
The authenticity of host '192.168.150.10' can't be established.
ED25519 key fingerprint is SHA256:FKV48qh9HK[…].
Are you sure you want to continue connecting (yes/no)?
```

The fingerprint is a hash value of the public host key, and once you confirm it with "yes," the host key will be added to *~/.ssh/known_hosts* and the host will then be considered known.

However, during the first connection attempt, you have no choice but to contact the administrator of the target system and verify the fingerprint. The administrator would then do something like the following on their system:

```
for file in /etc/ssh/*key.pub; do ssh-keygen -lf $file; done
256 SHA256:CCJ9m9wxsM[…] root@debian-12 (ECDSA)
256 SHA256:FKV48qh9HK[…] root@debian-12 (ED25519)
3072 SHA256:J6pcGyNQW8[…] root@debian-12 (RSA)
```


© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Remark

If the fingerprints are not listed in the desired format, you may need to add the `-E md5` option. This just depends on which format you want to compare with.

C.7.5 Restricted Keys

Occasionally, it's useful to restrict SSH access to a single command (especially for machine-based accesses). To achieve this, you can prepend a public key in the *authorized_keys* file with the `command="..."` option:

```
command="ls -l" ssh-rsa AAAAB3NzaC1yc2EAAAA[...]
```

Listing C.4 "authorized_keys": Example Restricted Key

With this key, you can just execute `ls -l` on this host—you don't even have to specify it when calling `ssh`.

If the command is to be parameterizable, then the `$SSH_ORIGINAL_COMMAND` variable is available inside `command`, and it contains all the parameters of the calling command line.

If you need to consider security aspects, then typically, you should use additional options:

```
command="...",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
```

Listing C.5 "authorized_keys"

C.7.6 Agent Forwarding

Using SSH agent forwarding is beneficial in situations where you need to "jump" from host to host (as shown in Figure C.2).

You can easily access Host A with a key and an agent, but you can't initially access Host B with the key because you'd have to upload your private key on Host A, which you obviously don't want to do. (Private means *private*!). The solution is so-called *agent forwarding*.

Functionality of Agent Forwarding

1. When establishing a connection to Host A, the client says, "I would like to use agent forwarding."
2. If Host A's configuration allows it, the `sshd` on Host A sets up an environment that can act like an SSH agent.
3. Then, when a connection to Host B is initiated, the client on Host A believes that an agent is running there (because `sshd` has "disguised" itself as an agent).

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

4. Then, `sshd` forwards the request back to the original machine. The agent there receives it and accesses your local key, and its response goes back to the `sshd` on Host A.
5. Then, the `sshd` on Host A passes the response to the SSH client, and the connection to Host B can be established.

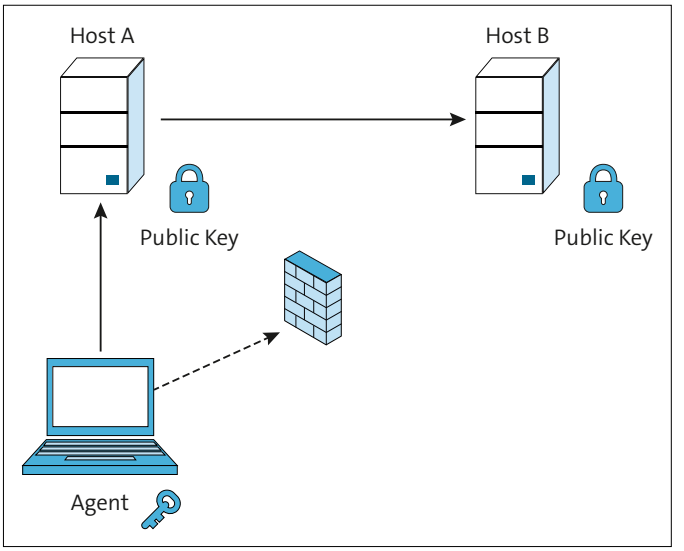


Figure C.2 SSH Agent Forwarding

Important

You should only use agent forwarding if you fully trust the intermediary host—because otherwise, an attacker could relatively easily intercept your private key.

Enabling Agent Forwarding

For the command line client, the two most common ways to enable agent forwarding are as follows:

1. You can do it with the `-A` option:

```
ssh -A <host_a>
```

2. You can do it with the SSH `ForwardAgent` client configuration directive:

```
Host host_a
    ForwardAgent yes
```

In a PuTTY session, you would need the following setting: **Connection • SSH • Auth, Allow agent forwarding.**

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Testing and Troubleshooting

When you test the setup, you should execute the `ssh-add -l` command on the first target host. This should display all keys that the agent has loaded on the source machine.

C.7.7 Transparent Jump Hosts

Here, we'll consider a scenario that's similar to the one in the previous section. However, this time, we'll aim for the jump host to remain completely unnoticed. Imagine, for example, that you want to copy a file from Host B to the source system *without* having to manually transport it from B to A and then from A to the source system.

The classic solution uses the SSH `ProxyCommand` directive. A prerequisite for convenient use is that you must have stored your public key on every system that's involved! Here is a typical configuration for such a scenario:

```
Host host_a
     Hostname ...
     Port ...
     User ...

Host host_b
     Hostname ...
     Port ...
     User ...
     ProxyCommand ssh -q -W %h:%p host_a
```

Listing C.6 "~/.ssh/config": Example Jump Host Scenario

You simply fill in your specific details, and then, you should be able to work directly with Host B without any interaction with jump host A. Without the SSH client configuration, the command would be a bit more cumbersome, as follows:

```
$ ssh -o ProxyCommand="ssh -q -W %h:%p <user@host_a>" <user@host_b>
```

At the very least, when nondefault ports are required, it will become no fun anymore. But now, with current OpenSSH versions, it's much shorter:

```
$ ssh -J <user@host_a> <user@host_b>
```

If you use nondefault ports, the notation will be `<user@host:port>`. A suitable SSH client configuration would look like this:

```
Host host_a
     Hostname ...
     Port ...
     User ...
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
Host host_b
     Hostname ...
     Port ...
     User ...
     ProxyJump host_a
```

Listing C.7 "~/.ssh/config": Example Jump Host Scenario

If you're faced with even more complex scenarios (e.g., a chain of jump hosts), I recommend you do further reading on at *https://en.wikibooks.org/wiki/OpenSSH/Cookbook/Proxies_and_Jump_Hosts*.

C.7.8 Escape Sequences

On the OpenSSH command line, there are a number of "secret" keyboard sequences called *SSH escape sequences*.

The most useful of these sequences by far is this one—it terminates the connection, and you'll mostly use it to terminate hanging connections:

Enter~.

There are (as mentioned) others; you can get an overview with this one:

Enter~?

C.8 "pssh" or "parallel-ssh"

In the *pssh* package, you will find useful tools to address many hosts simultaneously via SSH. (In Rocky/CentOS, the package is available in EPEL.)

Note

This section is only relevant for admins who don't feel like dealing with a tool like Ansible!

The typical procedure is as follows:

1. Set up a file that contains the access specifications of all target hosts, like the following:

   ```
   root@node1.example.org
   root@node2.example.org
   root@node3.example.org
   ```

   Listing C.8 "/etc/cluster-nodes"

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The name of this file is *not* fixed! The *cluster-nodes* name is just an example here, and you should choose any name that suits you!

2. I recommend that you set up an alias as follows:

```
#########
# Debian:
#########
alias pssh="parallel-ssh -h /etc/cluster-nodes -i"

##############
# SUSE, Rocky:
##############
alias pssh="pssh -h /etc/cluster-nodes -i"
```

Listing C.9 “/root/.bashrc”: Useful Alias for “parallel-ssh” or “pssh”

Next, you activate the settings by reloading the *.bashrc*:

```
$ source ~/.bashrc
```

Then, you can test the setup by doing something like running the `hostname` command on all nodes:

```
With the above alias:
$ pssh hostname
```

The *pssh* package also includes the following tools:

- **pscp**
  You can use this tool to simultaneously upload files.
- **prsync**
  You can use this tool to simultaneously synchronize files.
- **pnuke**
  You can use this took to simultaneously kill processes.

On Debian systems, the commands are called `parallel-*` accordingly.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Appendix D
Regular Expressions

Regular expressions (RegExes) are a powerful tool for processing text data.

You can utilize RegExes in various applications and services, including `grep`, `less`, `sed`, `awk`, `find`, `vi`, `emacs`, Ansible, Apache, and Postfix.

Of course, every common programming language also has a RegEx interface; they are most frequently used in scripting languages such as Perl and Python.

D.1 Motivation

Surely, you know the `grep` command, which you can use to search for certain text patterns. Suppose you want to "grep" to filter the output of `ls` so that only directories are listed:

```
$ ls -l | grep d
-rw-r--r-- 1 user1 users    6 Sep 26 11:53 data.txt
drwxr-xr-x 2 user1 users 4096 Sep 26 11:53 test1
drwxr-xr-x 2 user1 users 4096 Sep 26 11:53 test2
```

That doesn't work because all lines remain that contain a *d* anywhere. Instead, you are only interested in seeing the lines that have a *d* at the beginning of the line! You can do this by using the following search pattern:

```
$ ls -l | grep '^d'
drwxr-xr-x 2 user1 users 4096 Sep 26 11:53 test1
drwxr-xr-x 2 user1 users 4096 Sep 26 11:53 test2
```

Such a search pattern is a RegEx, and the additional caret symbol (^) is obviously not a normal character but a *metacharacter* with the special meaning "beginning of the line." The single quotes are actually unnecessary in this case, but you'll encounter some metacharacters that also have a meaning for the shell. These must then be quoted!

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

D.2 Dialects

Mastering RegExes is challenging for beginners because they need to learn numerous new symbols and their meanings and there are also multiple variants. Over the years, several "dialects" of RegExes have emerged with different syntax and capabilities.

Since I don't want to overcomplicate things, I'll just differentiate two of those dialects:

- **Perl Compatible RegExes (PCRE)**
  These are predominantly used in programming languages and modern tools and services (for example, in Ansible and Apache). They are based on the implementation in the Perl 5 programming language.
- **Portable Operating System Interface (POSIX) extended RegExes**
  These are used in more traditional tools like `grep`, `sed`, and `awk`.

A not insignificant part of the syntax is the same in all families, and we want to turn to this part first.

D.3 Basics

Table D.1 provides an overview of the basics, followed by examples.

Metacharacters	Meanings	Categories
`^`	Line start	Anchor symbol
`$`	Line end	Anchor symbol
`.`	Any character	Wildcard character
`\<C>`	Suppress special meaning of <C>	Quoting
`[...]`	One of the characters from "..."	Character class
`[^...]`	None of the characters from "..."	Character class
`<C>*`	Zero or more repetitions of <C>	Repetition Operator

Table D.1 RegExes, Basics

The classic `grep` program will serve as the testing tool. Its function should be familiar to you: only if the search pattern matches an input line will the line be printed to the output. However, in the following examples, I will state "`This matches`" or "`This does not match`" to avoid repeating the same output. And as I mentioned at the beginning, you should always enclose the RegEx in single quotes to prevent manipulation by the shell.

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Anchor Symbol

```
$ echo "Hello World" | grep '^He'
This matches.
$ echo "Hello World" | grep 'rld$'
This matches.
```

Wildcard Character

```
$ echo "Hello World" | grep 'el...Wo'
This matches.
```

Note that the space is also a character!

Quoting

```
$ echo "Please refer to point 2. in the document." | grep '2\.'
This matches.
$ echo "The CD contains 21 songs." | grep '2.'
This matches-but maybe not with our intent!
```

Character Classes

```
$ echo "Hello World" | grep '[Hh]ello'
This matches.
$ echo "Hello World" | grep 'H[a-e]ll[^a-e]'
This matches.
```

Range specifications like `a-e` are possible and typical, and an inverted character class like `[^a-e]` is best read as "none of the characters from a to e."

Repetition Operator

```
$ echo "Hello World" | grep 'He*Wo'
This does not match!
```

This time, I deliberately started with a negative example and a presumably incorrect use of the asterisk (`*`)character. The asterisk is not a wildcard but a symbol that in this case repeats the *preceding* `e`. For the regex engine, `He*Wo` means "Look for an `H`, followed by zero or more `e`, followed by a `W`, followed by an `o`."

```
$ echo "Hello World" | grep 'He.*Wo'
This matches.
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

This time, correctly and intentionally, the very typical `.*` signifies "zero or more of any character," and it acts somewhat like a wildcard for a string. With that in mind, you shouldn't be puzzled by this:

```
$ echo "Hello World" | grep 'HY*ello'
This matches. (There are zero Ys in it!)
```

D.4 Regular Expressions: Extended Capabilities (Perl Compatible Regular Expressions)

When discussing advanced matching possibilities, we need to distinguish the dialects, so let's start with the more modern PCRE dialect. First, see the overview in Table D.2.

Metacharacters	Meanings	Categories
`\d`, `\w`, `\s`	Digit, word character, and whitespace	Character class
`\D`, `\W`, `\S`	The respective opposite	Character class
`<C>+`	At least one <C>	Repetition operator
`<C>?`	Zero or one <C>	Repetition operator
`<C>{<N>}`	<N> times <C>	Repetition operator
`<C>{<M>,<N>}`	<M> to <N> times <C>	Repetition operator
`\b`	A word boundary	Anchor symbol
`<r1>|<r2>`	<r1> or <r2>	Alternative
`(...)`	"`...`" forms a logical unit	Grouping
`\1`, `\2` …	Match of the first, second, etc. bracket	Backreference

Table D.2 Regular Expressions, Extended Capabilities (PCRE)

As a testing tool, we continue to use `grep`, but now, we always use it with the `-P` option, which enables the PCRE syntax.

Tip

You're welcome to work with Perl (the original, so to speak). In most cases, it will actually deliver better performance. The recipe would look like this:

```
echo "Hello World" | perl -ne 'print if /^He.*ld$/'
```

Also note that the RegEx must be enclosed between two slashes, like this:

`/…/`

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Character Classes

```
$ echo "27 May 2024" | grep -P '\d\d\s\w\w\w\s\d\d\d\d'
This matches.
$ echo "I have 5 dogs and 7 cats" | grep -P '\d\D*\d'
This matches.
```

Read the second expression as "A digit, then many nondigits, and then another digit."

Repetition Operators

```
$ echo "Hello World" | grep -P 'Hel+'
This matches.

$ echo "Index file: index.htm" | grep -P 'index\.html?'
This matches (and the 'l' can be missing).

$ echo "Heeeello World" | grep -P 'e{4}'
This matches.
```

Anchor Symbol

```
$ echo "Hello World" | grep -P 'lo\b'
This matches. (The word ends after 'lo.')
$ echo "Hello World" | grep -P '\bell'
This does not match. (Before 'ell,' we are still in the middle of the word.)
```

Alternative

```
$ echo "Hello World" | grep -P 'jj|kk|ll|mm'
This matches.
```

Grouping

```
$ echo "A1B2C3D4" | grep -P '(\w\d){4}'
This matches (4x letter + digit).

$ echo "A1B2C3D4" | grep -P '\w\d{4}'
This does not match. (It's a letter followed by 4 digits.)
```

Backreference

```
$ echo "Hello World" | grep -P '(.)\1'
This matches. (A character repeats.)

$ echo "High noon" | grep -P '(.)(.)\2\1'
```


Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
This matches (and 'noon' is a palindrome).

$ echo "He said that that that was wrong." | grep -P '(\b\w+\b).*\1.*\1'
This matches.
```

The third expression checks whether a word (`\b\w+\b`) repeats twice more in the sequence.

Backreferences are among the most intriguing applications of regular expressions. However, they are more commonly used in search and replace operations, like in this example:

```
$ echo "The winning numbers are: 22, 27 and 49." | sed -r 's/([0-9]+)/#\1#/g'
The winning numbers are #22#, #27#, and #49#.
```

D.5 Extended Capabilities (Portable Operating System Interface Extended Regular Expressions)

Table D.3 shows the somewhat more classic extended RegExes.

Metacharacters	Meanings	Categories
<C>+	At least one <C>	Repetition operator
<C>?	Zero <C>s or one <C>	Repetition operator
<C>{<N>}	<N> times <C>	Repetition operator
<C>{<M>,<N>}	<M> to <N> times <C>	Repetition operator
\<	Word start	Anchor symbol
\>	Word end	Anchor symbol
<r1>|<r2>	<r1> or <r2>	Alternative
(...)	"..." forms a logical unit	Grouping
\1, \2 ...	Match of the first, second, etc. bracket	Backreference

Table D.3 RegExes: Extended Capabilities (POSIX Extended RegExes)

For example, you can test with these:

- `grep -E` or `egrep`
- `sed -r`
- `awk`

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

That would look like this, for example:

```
$ echo "Hello World" | grep -E '\<Wo'
This matches.

$ echo "Hello World" | sed -rn '/\<Wo/p'
This matches.

$ echo "Hello World" | awk '/\<Wo/ {print}'
This matches.
```

Well, almost—unless you have an `awk` version that doesn't quite implement the POSIX extended RegExes. The `GNU awk` version mostly does , but `mawk` does not. (The `awk -W version` provides information.)

In any case, I can assure you of this: if you experiment enough, you're bound to discover something that doesn't work as expected. For example, `awk` is a real challenge, and it also does not support backreferences. The implementation differences between the individual tools are undoubtedly present and sometimes very subtle.

The intention of this appendix was not to turn this topic into a scientific study, as previously stated. Jeffrey E. F. Friedl has already accomplished that in his excellent book *Mastering Regular Expressions* (O'Reilly Media, 2006), which I highly recommend that you read if your interest has been piqued!

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Appendix E
"vim" and "nano" Tips and Tricks

Terminal-based editors are essential in the Linux world, as many Linux installations are intentionally minimal. They lack graphical interfaces and, consequently, graphical editors and development environments.

This short appendix is intended to serve as your reference if you ever need to perform sophisticated text editing with an editor like `vim` or `nano`.

Important

You will find *no* basics in this appendix! I assume that you're familiar with basic operations such as editing text, saving, and exiting.

E.1 "vim"

In the system-wide */usr/share/vim/vimrc* file or a user-specific *~/.vimrc*, you can permanently set many configurations if you need to. Here are some useful examples:

```
" Enable syntax highlighting:
syntax on

" Jump to the last cursor position when opening a file:
au BufReadPost [...]
" (You can find this prepared in the global vimrc!)

" Constant display of the filename (possibly with full path):
set laststatus=2
" set statusline+=%F
```

Listing E.1 Useful Settings for "vim"

For several years, Debian has been applying some unusual default settings in the global *vimrc*. Notably, mouse support is enabled, and that interferes with standard text selection and pasting methods because it requires you to hold the `Shift` key. Additionally, the search function may seem odd if you're not accustomed to it.

If you want to remove this, you should uncomment the following line at the beginning of the global *vimrc* file:

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

```
let g:skip_defaults_vim = 1
```

Listing E.2 “/etc/vim/vimrc”: Excerpt

However, features like syntax highlighting and returning to the last known cursor position will be lost. The relevant lines are already set up, so if you need to, you can just uncomment them as well.

Table E.1 shows useful keyboard shortcuts and settings.

Keyboard Shortcuts	Effects
u or Ctrl+R	Undo or redo.
/<text>	Search.
n	Search next.
:set number	Toggle line numbering on or off.
:set list	Make nonprintable characters visible.
v or **V**, then cursor	Select an area.
dd or **d**	Cut the current line or selected area.
p or **P**	Paste a clipboard before or after current position.
< or **>**	Indent the selected area.
.	Repeat the last command.

Table E.1 Useful “vim” Commands

E.2 “nano”

In the system-wide */etc/nanorc* or a user-specific *~/.nanorc* file, you can permanently set many options if you need to. Here are some useful examples:

```
# Enable syntax highlighting:
include "/usr/share/nano/*.nanorc"

# Jump to the last cursor position when opening a file:
set positionlog

# Respond to Ctrl-Z:
set suspend
```

Listing E.3 Useful Settings for “nano”

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Table E.2 shows useful keyboard shortcuts and settings.

Keyboard Shortcuts	Effects
Alt+U or Alt+E	Undo or redo.
Ctrl+W	Search.
Alt+W	Search next.
Alt+C	Toggle display of cursor position (`set constantshow`).
Alt+# or Alt+N	Toggle line numbering (`set linenumbers`).
Alt+S	Toggle soft wrapping of overlong lines.
Alt+A, then cursor	Select an area.
Ctrl+K	Cut the current line or selected area.
Ctrl+U	Insert a clipboard.
Alt+{ or Alt+}	Indent the selected area.

Table E.2 Useful “nano” Keyboard Shortcuts

Tip

If a combination with the Alt key does not work as desired, then the Esc key is available as an alternative.

However, this is not a “hold key.” You must press it beforehand.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

The Author

Axel Miesen is an Ansible coach and has already introduced many admins to this useful tool. His interest in Linux systems began with his studies at the University of Kaiserslautern, where he studied mathematics and computer science. After graduating in 1998, he began working as a consultant and trainer and has been passing on his knowledge and experience of Linux topics as a consultant and trainer ever since.

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Index

.envrc 430

A

Active directory 346
Ad-hoc command 59
Adminer container 275
Agent forwarding 449
Ansible
  basic features 27
  core components 28
  history 30
  meaning 28
  updates 397
  versions 30
ansible_become 54
ansible_become_method 54
ansible_become_pass 54
ansible_become_user 54
ansible_env 167
ansible_password 49
ansible_user 49
ansible.cfg 46
  callbacks_enabled 353
  console_prompt 67
  cow_selection 259
  display_failed_stderr 352
  display_ok_hosts 352
  display_skipped_hosts 122, 352
  fact_caching 117
  force_handlers 98
  log_path 48
  nocows 258
  pipelining 257
  private_key_file 48
  retry_files_enabled 88
  retry_files_save_path 88
  roles_path 192
  show_custom_stats 352
  show_per_host_start 352
  show_task_path_on_failure 353
  stdout_callback 246, 352
  useful configuration 389
  vault_id_match 238
  vault_identity_list 242
ansible-bender 285
ansible-builder 226
ansible-cmdb 214
ansible-config 51
ansible-console 66
Ansible Galaxy 198
ansible-galaxy 197
ansible-inventory 289
ansible-lint 251
AnsibleModule object 368
ansible-playbook 90
ansible-pull 259
Ansible Vault 235
ansible-vault 237
apply 200
ARA 229
async 161
authorized_keys 441
Automation controller 223
AWS EC2 311
AWX 223
  execution environments 225

B

Basic configuration 46
Block 159
  exception handling 160

C

Callback plug-in 351
  implement 371
changed_when 113
Check mode 254
Code analysis
  static 251
Collections 169
  create 355
  listen 171
  manage 170
  search path 171
Colon 85
Configuration file
  INI 414
  XML 414
  YAML 413
Connection
  hanging 64
  persistent 63
Control host 32

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

cowsay ... 258
Custom module
  documentation ... 369
  parameter ... 361
  using Python ... 364

D

Debugging ... 244
delegate_facts ... 338
delegate_to ... 164, 335
Delegation ... 335
Diff mode ... 254
Directory structure ... 45
direnv ... 429
Docker ... 265
  build ... 283
  connection plug-in ... 282
  create image ... 283
  installation ... 265
  modules ... 269
Docker Compose ... 278

E

Environment variable ... 166
  ANSIBLE_COLLECTIONS_PATH ... 171
  ANSIBLE_CONFIG ... 46
  ANSIBLE_DEBUG ... 245
  ANSIBLE_DISPLAY_FAILED_STDERR ... 352
  ANSIBLE_DISPLAY_OK_HOSTS ... 352
  ANSIBLE_DISPLAY_SKIPPED_HOSTS ... 352
  ANSIBLE_ENABLE_TASK_DEBUGGER ... 249
  ANSIBLE_INVENTORY ... 57
  ANSIBLE_NOCOLOR ... 66
  ANSIBLE_PIPELINING ... 257
  ANSIBLE_SHOW_CUSTOM_STATS ... 352
  ANSIBLE_SHOW_PER_HOST_START ... 352
  ANSIBLE_SHOW_TASK_PATH_ON_FAILURE ... 353
  ANSIBLE_STDOUT_CALLBACK ... 246, 352
  ANSIBLE_VAULT_IDENTITY_LIST ... 241
  ANSIBLE_VERBOSITY ... 245
  DEFAULT_KEEP_REMOTE_FILES ... 245
Error handling ... 86, 158
Exception handling ... 160
Execution
  asynchronous ... 161
  conditional ... 121
  parallel ... 63
execution-environment.yml ... 226
Expression
  regular ... 455
Extra vars ... 104

F

Fact ... 115
  cache ... 117
facter ... 120
failed_when ... 158
File changes ... 254
Filter ... 137
  bool ... 138, 409
  default() ... 138, 154
  int ... 138
  join() ... 138, 403
  lower ... 138
  map() ... 405
  password_hash() ... 154, 408
  regex_replace() ... 403
  replace() ... 138
  selectattr() ... 404
  upper ... 138
force_handlers ... 98
Fork ... 63
Fully qualified collection name (FQCN) ... 355

G

Gateway host ... 401
gather_facts ... 81, 115
gather_subset ... 248
Git command ... 222
Gitea ... 219
GPG agent
  caching time ... 435
group_vars ... 106
Groups
  dynamic ... 295
  nested ... 290
GUI ... 213

H

Handler ... 95, 97
Hetzner Cloud ... 305
host_vars ... 108

I

Idempotency ... 61
ignore_errors ... 158

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

import_playbook ... 201
import_tasks ... 199
Imports ... 198
include_tasks ... 199
include_vars ... 201
Includes ... 198
Installation ... 36
  distribution packages ... 37
  PIP ... 38
Installer ... 391
Inventory ... 48
  alias ... 51
  dynamic ... 298
  group ... 49
  group parameterizing ... 49
  host parameterizing ... 49
  selection ... 56
  static ... 48
  YAML format ... 291
Inventory file ... 48
Inventory plug-in ... 301
  EC2 ... 315
  Hcloud ... 309
Inventory script ... 298
IP address ... 393

J

Jinja ... 131
  comment ... 136
  expression ... 136
  filter ... 137
  macro ... 140
  multiline statement ... 296
  statement ... 137
  test template ... 134
  whitespace control ... 138
Jumphost ... 401

K

Kerberos ... 346
Kuh ... 258

L

Lab environment ... 31
Link
  create ... 179
Linting ... 251
Linux updates ... 397
List ... 401
local_action ... 165
Locking ... 425
Logging ... 48
Lookup ... 143
Lookup plug-in ... 143
  csvfile ... 144
  dig ... 144
  env ... 144
  file ... 144
  fileglob ... 144
  filetree ... 144
  pipe ... 143
Loop ... 145
  over a block ... 424
loop_control ... 154
  index_var ... 156
  label ... 154
  loop_var ... 154

M

Map
  extend ... 401
  search ... 404
MariaDB container ... 275
max_fail_percentage ... 331
Meta
  end_play ... 85
  flush_handlers ... 98
Meta statements ... 186
Migration ... 43
Mode ... 178
Modularization ... 191
Module ... 359
  add_host ... 293
  apt ... 175
  assert ... 111, 189
  authorized_key ... 399
  blockinfile ... 180
  command ... 59, 112, 174
  copy ... 61, 177
  create ... 359
  cron ... 183
  debug ... 187, 256
  dnf ... 176
  docker_compose_v2 ... 274
  docker_container ... 270
  docker_container_exec ... 271
  docker_container_info ... 270
  docker_host_info ... 269
  docker_image ... 273
  docker_image_info ... 274

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

docker_login ... 272
Module (Cont.)
docker_network ... 271
docker_network_info ... 271
docker_prune ... 273
docker_volume ... 272
docker_volume_info ... 272
fail ... 187
file ... 61, 178
firewalld ... 184, 395
get_url ... 185
git ... 185
group ... 183
group_by ... 295
hostname ... 183
lineinfile ... 180
meta ... 186
package ... 61, 176
package_facts ... 176
pause ... 187
raw ... 174
reboot ... 61, 184
replace ... 181
script ... 175
service ... 182
service_facts ... 182
set_fact ... 189
shell ... 60, 112, 174
stat ... 179
template ... 178
unarchive ... 181
uri ... 185
user ... 61, 183
wait_for ... 188
wait_for_connection ... 188
win_chocolatey ... 349
win_domain_user ... 349
win_environment ... 350
win_regedit ... 350
win_updates ... 350
yum ... 176
zypper ... 176
module_defaults ... 410
Molecule ... 377
Ansible verifier ... 383
setup ... 377
Testinfra verifier ... 385

N

Notify ... 97

O

Ohai ... 120
Orchestration ... 325
Order ... 328
Ouput plug-in ... 246, 351

P

Packet version ... 296
parallel-ssh ... 452
pass ... 243, 433
Password
create ... 154, 408
generate ... 407
provide ... 400
random ... 407
Password manager ... 433
Patterns ... 67
Perl Compatible RegExes (PCRE) ... 458
ping ... 42
Ping test ... 48
Pipelining ... 257
Play ... 80
end ... 85
Playbook ... 79
import ... 201
lock ... 425
Playbook debugger ... 248
Play host ... 412
Play vars ... 103
Podman ... 287
Polemarch ... 231
poll ... 161
Portable Operating System Interface (POSIX) extended RegExes ... 460
post_tasks ... 195
pre_tasks ... 195
Privilege escalation ... 54
Prompting ... 109
Proxmox VE ... 318
Proxy ... 166
pssh ... 452
Pull mode ... 259
PuTTY ... 444

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

R

README.md 208
register 113
Regular expression 455
requirements.yml 211, 374
rescue 160
Retry files 88
Reusability 191
Role 191
create 197
dependencies 195
documentation 206
entry file 196
parametrize 194
reuse 209
search path 191
stopping 422
structure 192
run_once 164

S

scp 438
Semaphore 231
serial 329
set_fact 105
sftp 439
Software version 296
SSH 437, 438
agent 443
client configuration 448
escape sequences 452
jump host 449, 451
public key 41, 399
public key authentication 439
restricted key 449
ssh-add 443
ssh-agent 443
ssh-copy-id 442
SSH key
change passphrase 441
copy 442
create 440
fingerprint 448
ssh-keygen 440
ssh-keyscan 40
Stdout callback plug-ins 351
Strategy plug-ins 332
free 333
host_pinned 334
Strategy plug-ins (Cont.)
linear 332
Symlink 179

T

Tag 88
Target host 32
Task 83
Template 131
Test environment 31
throttle 328
Timeout 161
Tower 223
Troubleshooting 244

U

until 152
User 154

V

Vagrant 33
Vagrantfile 33
Variable 103
implicit 118
precedence rules 105
precedences 104
Variable structure 110
vars 103
vars_files 109
vars_prompt 109
Vault 235
ID 237
Version number 122
Virtualenv environment 39

W

Web interface 213
when 121
Windows module 348
Windows target 339
WinRM 343
setup 344
troubleshooting 345
WinSCP 446
with_dict 149
with_fileglob 146
with_first_found 201
with_items 146

Personal Copy for Jaleel Hussain, alex76alex43@gmail.com

with_list 148
with_nested 150
with_sequence 149
with_subelements 151
with_together 151

Y

YAML 69
anchor 77
block expression 75
YAML (Cont.)
list 73
map 73
nested structures 73
NULL value 76
reference 77
syntax 69

© 2025 by Rheinwerk Publishing Inc., Boston (MA)

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: rachelg@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also share your reading experience via Twitter, Facebook, or email.

Supplements

If there are supplements available (sample code, exercise materials, lists, and so on), they will be provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: http://www.rheinwerk-computing.com/6068. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at Rheinwerk Computing, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website http://www.rheinwerk-computing.com provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at http://www.rheinwerk-computing.com.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

This publication is protected by copyright in its entirety. All usage and exploitation rights are reserved by the author and Rheinwerk Publishing; in particular the right of reproduction and the right of distribution, be it in printed or electronic form.

© 2025 by Rheinwerk Publishing, Inc., Boston (MA)

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you store this copy on a device that is solely and personally used by yourself. You are not entitled to any other usage or exploitation.

In particular, it is not permitted to forward electronic or printed copies to third parties. Furthermore, it is not permitted to distribute the e-book on the Internet, in intranets, or in any other way or make it available to third parties. Any public exhibition, other publication, or any reproduction of the e-book beyond personal use are expressly prohibited. The aforementioned does not only apply to the e-book in its entirety but also to parts thereof (e.g., charts, pictures, tables, sections of text).

Copyright notes, brands, and other legal reservations as well as the digital watermark may not be removed from the e-book.

No part of this book may be used or reproduced in any manner for the purpose of training artificial intelligence technologies or systems. In accordance with Article 4(3) of the Digital Single Market Directive 2019/790, Rheinwerk Publishing, Inc. expressly reserves this work from text and data mining.

Digital Watermark

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy. If you, dear reader, are not this person, you are violating the copyright. So please refrain from using this e-book and inform us about this violation. A brief email to info@rheinwerk-publishing.com is sufficient. Thank you!

Trademarks

The common names, trade names, descriptions of goods, and so on used in this publication may be trademarks without special identification and subject to legal regulations as such.

All products mentioned in this book are registered or unregistered trademarks of their respective companies.

Limitation of Liability

Regardless of the care that has been taken in creating texts, figures, and programs, neither the publisher nor the author, editor, or translator assume any legal responsibility or any liability for possible errors and their consequences.