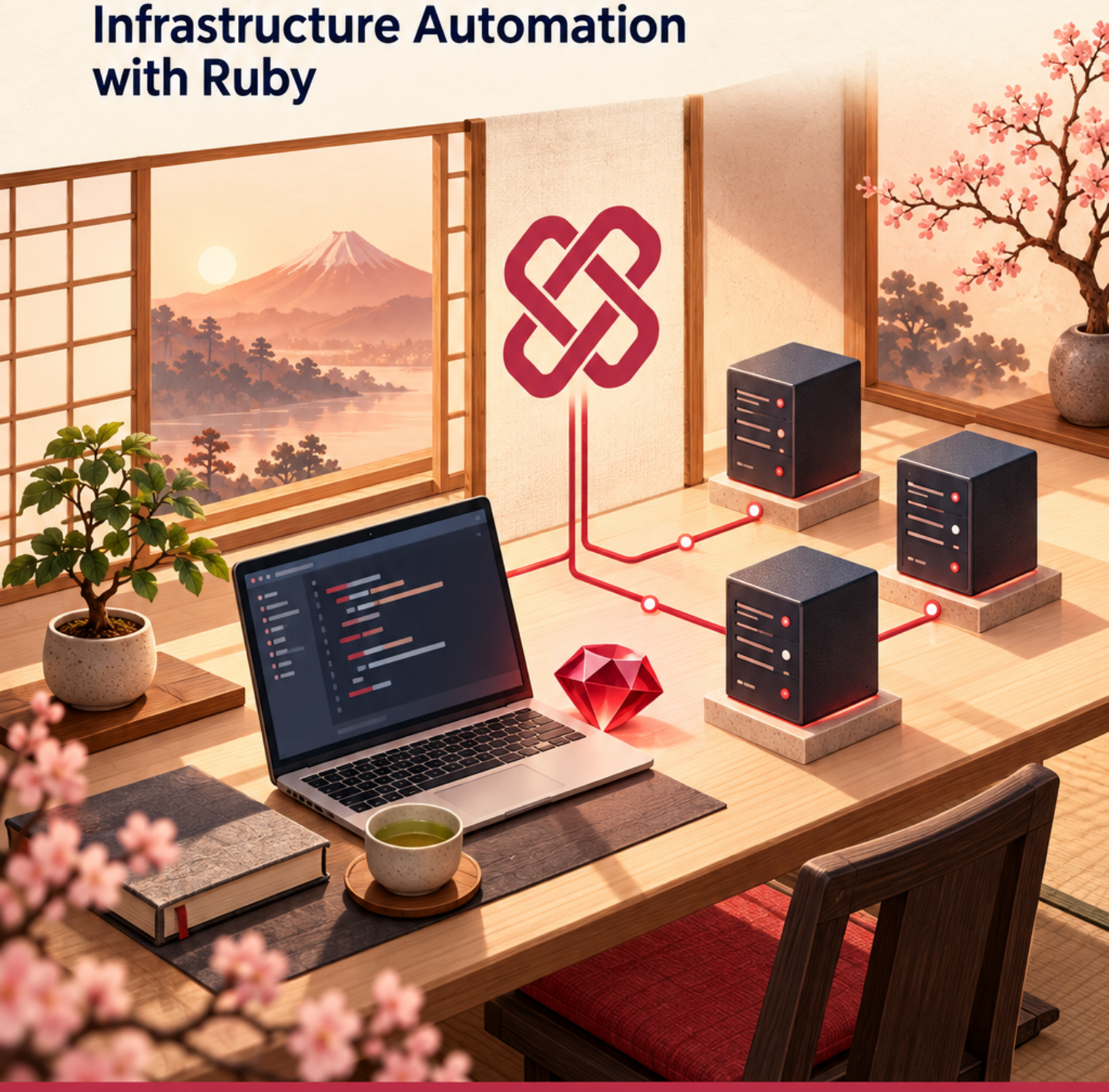
Sadame:
Configuration Management Made Simple

The Beginner's Guide to Infrastructure Automation with Ruby



Sadame: Configuration Management Made Simple

The Beginner's Guide to Infrastructure Automation with Ruby

Ender KUŞ

Version 1.0, November 2026

Contents

Colophon . . . **1**
Preface . . . **2**
Who this book is for . . . 2
What you will build . . . 2
The practice lab . . . 3
Conventions . . . 3
Getting the code . . . 3
I: First Steps . . . 4
1. Why Configuration Management? . . . **4**
1.1. The server nobody can rebuild . . . 4
1.2. Scripts help, a little . . . 4
1.3. Describe the result instead . . . 4
1.4. Plan, apply, drift . . . 5
1.5. How sadame reaches your servers . . . 5
1.6. Where sadame fits . . . 6
2. Setting Up Your Lab . . . **7**
2.1. What is in the lab . . . 7
2.2. What you need . . . 7
2.3. Starting the lab . . . 7
2.4. Three scripts, all you need . . . 8
2.5. Installing sadame on your own computer . . . 9
3. Your First Recipe . . . **10**
3.1. Two lines of Ruby . . . 10
3.2. Plan: what would change? . . . 10
3.3. Apply: make it so . . . 11
3.4. Apply again: nothing to do . . . 11
3.5. Drift: someone changed it . . . 12
3.6. Repair . . . 12
3.7. Removing things . . . 13
3.8. Output for programs . . . 13
4. Just Enough Ruby . . . **15**
4.1. A recipe is a program . . . 15
4.2. Strings . . . 16
4.3. Variables . . . 16
4.4. Symbols . . . 16
4.5. Arrays and loops . . . 17
4.6. Hashes . . . 17
4.7. Conditions . . . 17
4.8. Methods . . . 17
4.9. Ruby runs during plan too . . . 18
II: The Building Blocks . . . 19
5. Packages and Files . . . **19**
5.1. Packages . . . 19
5.2. Files with content . . . 20
5.3. Permissions drift too . . . 22
5.4. How sadame writes a file . . . 22
5.5. Check before you replace . . . 22
6. Templates and Variables . . . **25**
6.1. vars.rb: the values in one place . . . 25

6.2. Templates . . . 25
6.3. locals: values for one template . . . 26
6.4. Changing a value from the command line . . . 27
6.5. Where vars come from . . . 27
6.6. When a template is wrong . . . 27
7. Directories, Links and Lines . . . 29
7.1. Directories . . . 30
7.2. Links . . . 30
7.3. Lines . . . 31
7.4. Which resource for a file? . . . 31
8. Services and Notifications . . . 33
8.1. The service resource . . . 33
8.2. Notifications . . . 33
8.3. A change triggers the reload . . . 35
8.4. Services drift too . . . 35
8.5. Actions and timing . . . 36
9. Users, Groups and Commands . . . 38
9.1. Groups and users . . . 39
9.2. Commands with exec . . . 39
9.3. Commands that run only when notified . . . 40
9.4. Nothing left to do . . . 41
III: Many Machines . . . 43
10. Reaching Servers over SSH . . . 43
10.1. One option makes it remote . . . 43
10.2. Permission to change things . . . 44
10.3. Several servers at once . . . 45
10.4. What happens on the server . . . 46
11. The Inventory . . . 47
11.1. hosts.rb . . . 47
11.2. Seeing what the inventory says . . . 47
11.3. Choosing hosts . . . 48
11.4. All of them . . . 49
11.5. Overriding a var once . . . 50
11.6. Inventories from somewhere else . . . 51
12. Facts and Platforms . . . 52
12.1. What a machine looks like . . . 52
12.2. The platform . . . 53
12.3. One recipe, two families . . . 53
12.4. Facts follow the machine . . . 55
12.5. When the platform is not supported . . . 55
13. Organizing a Growing Recipe . . . 56
13.1. The layout . . . 56
13.2. include_recipe . . . 56
13.3. Tags . . . 57
13.4. Starting a project with sadame init . . . 59
IV: Projects . . . 61
14. Project: A Website . . . 61
14.1. How Debian's nginx finds sites . . . 61
14.2. The recipe . . . 61
14.3. Plan, apply, look . . . 62
14.4. Changing the site . . . 64

14.5. Going further . . . 64
15. Project: A Database Server . . . 66
15.1. The plan for the recipe . . . 66
15.2. The recipe . . . 66
15.3. Tuning from facts . . . 67
15.4. Keeping the password out . . . 68
15.5. Running it . . . 68
15.6. Does it work? . . . 70
16. Project: Deploying an Application . . . 72
16.1. The application . . . 72
16.2. The recipe . . . 72
16.3. The first deployment . . . 73
16.4. Releasing version 1.1 . . . 75
16.5. One server at a time . . . 75
17. Project: The Whole Fleet . . . 78
17.1. The shape of it . . . 78
17.2. A new role: hardening . . . 79
17.3. Check before you build . . . 80
17.4. From empty servers to a fleet . . . 80
17.5. Running one role . . . 81
17.6. Where to go from here . . . 82
V: Day Two . . . 83
18. Drift Detection and CI . . . 83
18.1. Three small accidents . . . 83
18.2. Reports for people . . . 84
18.3. Repairing . . . 85
18.4. A daily check in CI . . . 86
19. Safety Nets . . . 88
19.1. Layers of safety . . . 88
19.2. validate . . . 88
19.3. verify: the last check . . . 89
19.4. The editor . . . 90
19.5. Habits . . . 91
20. When Things Go Wrong . . . 93
20.1. Three kinds of failure . . . 93
20.2. A misspelled option . . . 93
20.3. A notification to nowhere . . . 94
20.4. A template error . . . 94
20.5. A package that does not exist . . . 94
20.6. Hosts that cannot be reached . . . 95
20.7. A checklist . . . 96
21. Where to Go Next . . . 98
21.1. From the lab to your servers . . . 98
21.2. Working as a team . . . 98
21.3. Sharing recipes . . . 99
21.4. Coming from another tool . . . 99
21.5. Your own computer . . . 99
21.6. When sadame is not the right tool . . . 99
21.7. Keep learning . . . 100
21.8. A last word . . . 100
Appendix A: Command Reference . . . 101

A.1. Synopsis . . . 101
A.2. Commands . . . 101
A.3. Choosing hosts . . . 101
A.4. Other options . . . 102
A.5. Exit status . . . 102
A.6. In a recipe . . . 103
A.7. In hosts.rb . . . 103
Appendix B: Resource Reference . . . 104
B.1. file . . . 104
B.2. directory . . . 104
B.3. link . . . 104
B.4. line . . . 105
B.5. package . . . 105
B.6. service . . . 105
B.7. user . . . 105
B.8. group . . . 106
B.9. exec . . . 106
B.10. git . . . 106
B.11. Notifications . . . 106
Appendix C: Facts and Platforms . . . 108
C.1. Facts . . . 108
C.2. Platform . . . 108
C.3. Names that differ between families . . . 109
Appendix D: Glossary . . . 110
Index . . . 112

Colophon

Sadame: Configuration Management Made Simple. The Beginner's Guide to Infrastructure Automation with Ruby.

Copyright © 2026 Ender KUŞ.

The text of this book is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0). The code in its repository is licensed under the MIT License. Both are at https://github.com/enderkus/sadame-beginner-guide.

First edition, November 2026, written for sadame 0.5.1. Every command and every output in this book was run and captured in the book's practice lab with that version.

Read online at https://enderkus.github.io/sadame-beginner-guide/, where the PDF and EPUB editions can also be downloaded.

Preface

Every server starts clean. Then someone installs a package, edits a configuration file, adds a user for a colleague, fixes something at two in the morning. A year later nobody can say what is on the machine, why, or how to build another one like it.

Configuration management is the habit, and the tooling, that prevents this: you write down what a machine should look like, and a tool makes it so, shows you every change before it happens, and tells you when the machine has wandered away from what you wrote. This book teaches that habit with **sadame**, a small tool that keeps the whole idea within reach: the descriptions are plain Ruby, every change is a diff you can read, and the tool is a single file that needs nothing installed on your servers.

Who this book is for

You know your way around a Linux shell: you can `cd`, edit a file and run a command with `sudo`. You may have looked after a server or two by hand, or be about to. You do not need to know Ruby, nor any other configuration management tool; chapter 4 teaches all the Ruby the book uses, and chapter 1 explains the ideas from the start.

If you already use Ansible, Chef, Puppet or Itamae, the early chapters will go quickly, and chapter 21 maps what you know onto sadame.

What you will build

The book starts with a recipe of two lines and ends with a small fleet: two web servers behind nginx running an application deployed from git, a PostgreSQL server tuned to its machine, every server hardened, all of it rolled out one server at a time, checked every day for drift, and checked for mistakes before it reaches a server at all.

Part I, First Steps	why configuration management exists, a practice lab on your own computer, your first recipe, and just enough Ruby.
Part II, The Building Blocks	the resources you describe machines with: packages, files, templates, directories, links, single lines, services, users, groups and commands.
Part III, Many Machines	reaching servers over SSH, describing them in an inventory, adapting recipes to each machine, and organizing recipes as they grow.
Part IV, Projects	a website, a database server, an application deployed from git, and finally all of them together.
Part V, Day Two	drift detection and CI, the safety nets that catch mistakes early, troubleshooting, and where to go next.

The appendices hold quick references for the command line, every resource and every fact, and a glossary.

The practice lab

Everything in this book runs in a lab of four small machines on your own computer, built from the book's repository with Docker: one you work on, and three servers it manages. You can break them, throw them away and start again in a minute. Chapter 2 sets the lab up; you need Docker and about two gigabytes of free disk space.

All the output in the book comes from that lab. When you run the same commands, yours will match, apart from timings, a few numbers that come from your computer (how many CPUs and how much memory it has) and the order in which lines from different servers arrive.

Conventions

Commands you type are shown after a `$` prompt, followed by their output:

```
$ sadame --version
sadame 0.5.1
```

Recipes and other files are shown with their path in the book's repository as the title. `code/03-first-recipe/hello.rb` is in the `code` directory of the repository, and in `~/code` on the lab's practice machine.



Notes like this one add detail you can skip on a first reading.



Tips suggest a better way, or a shortcut.



Warnings mark the places where a mistake costs something.

Every chapter ends with a short **Try it** section: exercises that use what the chapter showed. They have no answers at the back; the lab and `sadame plan` tell you whether you got them right.

Getting the code

The book's repository holds every recipe, the lab, and the sources of the book itself:

```
$ git clone https://github.com/enderkus/sadame-beginner-guide.git
```

Found a mistake, or something unclear? Open an issue at https://github.com/enderkus/sadame-beginner-guide.

Part I: First Steps

Chapter 1. Why Configuration Management?

Before any tool, a story you may recognize. It explains every idea the rest of the book uses: desired state, idempotence, plan, apply and drift.

1.1. The server nobody can rebuild

A small company rents a server and installs nginx on it to serve its website. Over the next months the server collects changes: a package for image resizing, a tweak to nginx for larger uploads, a user account for the contractor who left last spring, a cron job that someone added "temporarily", a firewall rule copied from a blog post. Each change was made by hand, over SSH, by whoever was on call.

Then the disk dies, or the provider retires the machine, or the company needs a second server for the traffic. Now someone has to build a machine like the old one, and the only description of the old one is the old one. Snowflake servers like this are unique in the worst sense: you cannot reproduce them, you cannot tell what is on them, and you cannot change them without fear.

1.2. Scripts help, a little

The first fix most teams reach for is a script: put the commands in a file and run it on the new machine.

```
apt-get install -y nginx
echo "client_max_body_size 20m;" >> /etc/nginx/conf.d/uploads.conf
useradd deploy
systemctl enable --now nginx
```

The script is a description of the server, which is a big step up. But it describes **how to get there from an empty machine**, once. Run it a second time and `useradd` fails because the user exists, and the configuration file grows a second copy of the line. Run it on a machine that is half configured and nobody knows what happens. A script records actions; what you want to record is a result.

1.3. Describe the result instead

Configuration management turns this around. You write down the desired state of the machine, what should be true when you are done, and the tool works out the actions:

```
package "nginx"
file "/etc/nginx/conf.d/uploads.conf", content: "client_max_body_size 20m;\n"
user "deploy"
service "nginx", enable: true, state: :running
```

This is a sadame recipe. Each line is a resource: one thing about the machine, with the state it

should be in. nginx installed. That file, with exactly that content. That user present. That service running, and started at boot.

For each resource, the tool asks the machine how things are now, compares that with the recipe, and changes only what differs. If nginx is already installed, it leaves it alone. If the file has the right content, it does not touch it. If someone edited the file, it puts the right content back.

This property has a name: idempotence. Applying a recipe once or ten times leaves the machine in the same state, and applying it to a machine that is already right changes nothing. It is what makes a recipe safe to run at any time, on any machine, as often as you like.

1.4. Plan, apply, drift

Sadame gives you three commands built on that comparison, and you will use them in every chapter:

`sadame plan`

: Compares the machine with the recipe and prints what would change, as a diff, without changing anything. You read it before you act.

`sadame apply`

: Makes the same comparison and changes what differs, one resource after another.

`sadame drift`

: Makes the same comparison and answers one question with its exit status: does the machine still match the recipe? It is meant for scripts and for CI, which can run it every morning and tell you when someone changed a server by hand.

That last idea, configuration drift, is the slow wandering of a machine away from its description: the hand edits, the hotfixes, the "temporary" changes. A recipe that you apply once and never check again describes the machine on the day you applied it. A recipe that you check for drift every day describes the machine today.

1.5. How sadame reaches your servers

Some configuration management tools need software installed and running on every server, an agent, that fetches recipes from a central server. Sadame does not. It works in push mode: from your own computer, over the same SSH you use already, it copies itself to each server for the duration of a run, sends the recipe along, runs it there, and cleans up. Nothing keeps running afterwards.

Sadame itself is a single file of about one and a half megabytes. The recipes are Ruby, but Ruby is built into that file, so neither your computer nor your servers need Ruby installed. The servers need SSH, a shell and `tar`, which every Linux server has.

1.6. Where sadame fits

Sadame is not the only tool of its kind, and it is honest about being young. Ansible, Chef, Puppet and Itamae have run servers for years and can do far more. Sadame aims at a smaller, clear target:

- recipes in **plain Ruby**, so conditions, loops and helper methods are just Ruby, not a templating language on top of YAML;
- a **diff before every change**, for every file, package and service;
- **nothing to install on the servers**, and one small binary on your side;
- a **drift check built for CI**, with reports for GitHub and JUnit;
- a design small enough to learn in a weekend, which is what this book sets out to show.

Sadame runs on Linux and macOS and manages Debian, Ubuntu, Rocky Linux and other RHEL-like systems, Alpine, and Arch Linux, on x86_64 and ARM64.

What you learned

- Hand-configured servers drift into states nobody can describe or rebuild.
- Scripts record actions; configuration management records the desired state and works out the actions.
- A recipe is made of resources, and applying it is idempotent.
- `plan` shows the changes, `apply` makes them, `drift` tells you whether a machine still matches.
- Sadame pushes itself over SSH; servers need nothing installed.

Try it

1. Think of a machine you look after, or one at work. Write down, in plain English, five things that must be true about it: packages, files, users, services. That list is the first draft of a recipe.
2. For each item, ask: if someone changed it by hand tomorrow, how would you find out? Chapter 18 answers that for every item at once.

Chapter 2. Setting Up Your Lab

You learn configuration management by changing machines, which is exactly what you should not practice on machines that matter. This chapter builds a lab on your own computer: four small Linux machines in Docker containers that behave like real servers, that you can break on purpose and replace in a minute.

2.1. What is in the lab

Machine	System	Role in the book
`practice`	Debian 12	The machine you work on. Sadame is installed here, and so is your copy of the book's code, in `~/code`. Chapters 3 to 9 change this machine itself; from chapter 10 on it manages the servers over SSH.
`web1`, `web2`	Debian 12	Two web servers.
`db1`	Rocky Linux 9	A database server, deliberately a different distribution, so you see how one recipe handles two families of Linux.

Each machine boots **systemd**, the service manager real servers use, so services start and stop for real. The servers have a user called `deploy` who may use `sudo` without a password, as many cloud servers are set up, and they accept SSH logins with the practice machine's key only.

2.2. What you need

- **Docker**: Docker Desktop on macOS or Windows (with WSL 2), OrbStack on macOS, or Docker Engine on Linux. The lab uses `docker compose`, which all of them include.
- **Git**, to get the book's repository.
- About **two gigabytes** of disk space and a working internet connection: the lab downloads Debian and Rocky Linux images, and packages as you install them.



The containers run with the `privileged` flag, which systemd needs inside a container. That is fine for a lab on your own computer; it is not how you would run containers in production.

2.3. Starting the lab

Get the book's repository and start the lab with its script:

```
$ git clone https://github.com/enderkus/sadame-beginner-guide.git
$ cd sadame-beginner-guide
$ lab/up.sh
```

The first start builds the four machines' images, which takes a few minutes; later starts take seconds. The script waits until every machine has booted, gives the practice machine an SSH key, and lets the servers accept it. When it is done, it says so:

```
$ lab/up.sh
waiting for practice
waiting for web1
waiting for web2
waiting for db1 .
The lab is ready. Enter the practice machine with: lab/shell.sh
```

Now step into the practice machine:

```
$ lab/shell.sh
```

Your prompt changes to `ops@practice`: you are the user `ops` on the practice machine, in your copy of the code. Check that sadame is there, and look at the code:

```
$ sadame --version
sadame 0.5.1

$ ls
03-first-recipe
04-ruby
05-packages-and-files
06-templates-and-vars
07-directories-links-lines
08-services-and-notifications
09-users-groups-commands
10-remote-hosts
11-inventory
12-facts-and-platforms
13-organizing
14-website
15-database
16-application
17-fleet
18-drift-and-ci
19-safety-nets
20-troubleshooting
README.md
```

There is one directory per chapter. From now on, every command in the book runs here, on the practice machine, unless the text says otherwise.

2.4. Three scripts, all you need

Script	What it does
`lab/up.sh`	Starts the lab, or makes sure it is running. Safe to run any time.
`lab/shell.sh`	Opens a shell on the practice machine as `ops`, in `~/code`.
`lab/down.sh`	Stops the lab and throws its machines away. The next `lab/up.sh` starts from fresh ones, as if nothing had happened.

That last one is your undo button. If a server ends up in a state you do not understand, `lab/down.sh` followed by `lab/up.sh` gives you a clean lab. Your copy of the code goes with it, so keep anything you want to keep in the repository on your own computer.



Chapters build on each other: chapter 8 expects what chapter 5 installed, and the projects in Part IV expect the servers to be reachable. If you start reading in the middle, the chapter's commands still work on a fresh lab, but the outputs will show more changes than the book's.

2.5. Installing sadame on your own computer

The lab is all you need for this book. When you start using sadame for real, install it on the computer you manage your servers from:

```
$ curl -fsSL https://raw.githubusercontent.com/enderkus/sadame/main/install.sh | sh
```

The script picks the right binary for your system, checks its SHA-256 checksum against the release, and installs it to `/usr/local/bin`. On macOS and Linux with Homebrew, `brew install enderkus/sadame/sadame` does the same. Either way, `sadame --version` should print the version.

The servers you manage need nothing: sadame brings itself along.

What you learned

- The lab has a practice machine and three servers: two Debian, one Rocky Linux.
- `lab/up.sh` starts it, `lab/shell.sh` takes you in, `lab/down.sh` resets everything.
- On your own computer, sadame is one file, installed by a script or Homebrew.

Try it

1. On the practice machine, run `ssh web1 hostname` and `ssh db1 cat /etc/os-release`. You are logging in as `deploy`, with the key the lab set up.
2. Run `lab/down.sh` and `lab/up.sh` from your own computer, and time the second start. That is how long a fresh start takes when you need one.

Chapter 3. Your First Recipe

In this chapter you write a recipe of one line, and use it to see the whole cycle sadame is built on: plan a change, apply it, apply it again and see nothing happen, break the machine by hand, catch it, and repair it.

3.1. Two lines of Ruby

Go to the chapter's directory on the practice machine and look at the recipe:

```
$ cd ~/code/03-first-recipe
```

code/03-first-recipe/hello.rb

```
# Your first recipe: one file, with fixed content.
file "/tmp/hello.txt", content: "Hello from sadame!\n"
```

The first line is a Ruby comment. The second declares one resource: a **file** at `/tmp/hello.txt` whose content is the text `Hello from sadame!` followed by a newline. Every resource has the same three parts:

Part	Here	What it is
Type	`file`	What kind of thing it is: a file, a package, a service…
Name	`"/tmp/hello.txt"`	Which one. For a file, its path.
Options	`content: "Hello from sadame!\n"`	What it should be like.

Sadame writes a resource as `type[name]`, so this one is `file[/tmp/hello.txt]`. You will see that form in every output.



The file lives in `/tmp` so that you can run this chapter as an ordinary user. Recipes that change the system, from chapter 5 on, run with `sudo`.

3.2. Plan: what would change?

Ask sadame what applying the recipe would do:

```
$ sadame plan hello.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

+ file[/tmp/hello.txt]
    state: absent -> present
    --- /dev/null
    +++ /tmp/hello.txt (desired)
    @@ -0,0 +1 @@
    +Hello from sadame!

Plan: 1 to create, 0 to update, 0 to remove, 0 to run, 0 unchanged.
```

Read the output from the top:

- The first line says what sadame is doing (`plan`), where (`local`, this machine) and on what system.
- `+ file[/tmp/hello.txt]` means the file would be **created**. A `~` would mean **updated**, a `-` **removed**.
- `state: absent → present` is the change in short: the file does not exist and should.
- The lines after it are a unified diff, the format `git diff` uses, between the file as it is (`/dev/null`, nothing) and as the recipe wants it. Lines starting with `+` would be added.
- The last line counts the resources by what would happen to them.

`plan` changed nothing. Check: `ls /tmp/hello.txt` still finds nothing. That is the point of `plan`: you can run it whenever you want, on any machine, as often as you like, to see what the recipe would do.

3.3. Apply: make it so

Now let sadame make the change:

```
$ sadame apply hello.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

changed    file[/tmp/hello.txt]
           state: absent -> present
           --- /dev/null
           +++ /tmp/hello.txt (desired)
           @@ -0,0 +1 @@
           +Hello from sadame!

Summary
  changed      1  file[/tmp/hello.txt]
  unchanged    0
  failed       0
  skipped      0
  notified     0

Finished in 0.00s.

$ cat /tmp/hello.txt
Hello from sadame!
```

The resource is marked `changed`, with the same diff the plan showed, and a summary counts the results. The file is there, with the recipe's content.

3.4. Apply again: nothing to do

Run the plan once more:

```
$ sadame plan hello.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

Plan: 0 to create, 0 to update, 0 to remove, 0 to run, 1 unchanged.
```

The file already matches the recipe, so there is nothing to do: one resource unchanged. Running `apply` now would also change nothing. This is idempotence at work, and it is what makes recipes safe: you never have to remember whether you applied one already.

3.5. Drift: someone changed it

Behave like a colleague in a hurry and edit the file by hand, then ask sadame whether the machine still matches the recipe:

```
$ echo 'Hello from someone else' > /tmp/hello.txt

$ sadame drift hello.rb
sadame 0.5.1 drift on local (Debian GNU/Linux 12 (bookworm))

~ file[/tmp/hello.txt]
    --- /tmp/hello.txt
    +++ /tmp/hello.txt (desired)
    @@ -1 +1 @@
    -Hello from someone else
    +Hello from sadame!

Drift: 1 of 1 resource differs from the recipe.

$ echo $?
1
```

`drift` makes the same comparison as `plan`, and prints the same kind of diff: the line starting with `-` is what is on the machine, the line with `+` what the recipe wants. What makes `drift` different is its exit status: `1` because the machine differs from the recipe. A script or a CI job can act on that number without reading the output.

3.6. Repair

Apply the recipe again, and check:

```
$ sadame apply hello.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

changed    file[/tmp/hello.txt]
           --- /tmp/hello.txt
           +++ /tmp/hello.txt (desired)
           @@ -1 +1 @@
           -Hello from someone else
           +Hello from sadame!

Summary
  changed      1  file[/tmp/hello.txt]
  unchanged    0
  failed       0
  skipped      0
  notified     0

Finished in 0.00s.

$ sadame drift hello.rb
sadame 0.5.1 drift on local (Debian GNU/Linux 12 (bookworm))

No drift: 1 resource matches the recipe.

$ echo $?
0
```

The hand edit is gone, and `drift` exits with `0`: no drift. Plan, apply and drift are three views of the same comparison:

Command	Changes the machine	Exit status
`plan`	no	`0` when the plan could be computed, whether or not there are changes
`apply`	yes	`0` when every resource succeeded, `1` when one failed
`drift`	no	`0` when the machine matches, `1` when it differs

All three exit with `2` when they could not do their job at all: a typo in the recipe, a machine that cannot be reached. Appendix A lists every exit status.

3.7. Removing things

A recipe can also say that something must **not** exist:

code/03-first-recipe/goodbye.rb

```
# The same file, declared absent: sadame removes it if it is there.
file "/tmp/hello.txt", state: :absent
```

```
$ sadame apply goodbye.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

changed    file[/tmp/hello.txt]
           state: present -> absent

Summary
  changed      1  file[/tmp/hello.txt]
  unchanged    0
  failed       0
  skipped      0
  notified     0

Finished in 0.00s.

$ ls /tmp/hello.txt
ls: cannot access '/tmp/hello.txt': No such file or directory
```

`state: :absent` works for most resources: files, directories, packages, users, groups. Removing something from a recipe does **not** remove it from the machine; sadame only manages what the recipe declares. To remove something, declare it absent.

3.8. Output for programs

Every command can print its result as JSON instead of text, for scripts and other tools:

```
$ sadame plan hello.rb --json
{
  "format_version": 1,
  "sadame_version": "0.5.1",
  "command": "plan",
  "host": "local",
  "platform": {
    "name": "Debian GNU/Linux 12 (bookworm)",
    "family": "debian",
    "package_manager": "apt"
  },
  "tags": [],
```

```
  "skip_tags": [],
  "resources": [
    {
      "id": "file[/tmp/hello.txt]",
      "type": "file",
      "name": "/tmp/hello.txt",
      "status": "unchanged",
      "changes": [],
      "notifications": [],
      "error": null
    }
  ],
  "notifications": [],
  "summary": {
    "create": 0,
    "update": 0,
    "remove": 0,
    "run": 0,
    "unchanged": 1,
    "failed": 0
  },
  "drift": false,
  "failed": false,
  "exit_status": 0,
  "duration_seconds": 0.0
}
```

Each resource has its `id`, its `status` and its `changes`; the document ends with the counts and the exit status. Chapter 18 turns this JSON into reports for CI.

What you learned

- A resource is a type, a name and options; sadame writes it `type[name]`.
- `plan` shows the changes as diffs and changes nothing.
- `apply` changes only what differs; applying again changes nothing.
- `drift` exits with `1` when the machine differs from the recipe.
- `state: :absent` removes things; deleting a resource from the recipe does not.
- `--json` prints any result as JSON.

Try it

1. Change the content in `hello.rb` to two lines, `"Hello\nWorld\n"`, and run `plan`. How does the diff show two lines?
2. Add a second resource to `hello.rb` for `/tmp/second.txt`, then `plan` and `apply`. What do the summary counts say?
3. Run `chmod 0600 /tmp/hello.txt`, then `sadame plan hello.rb`. Nothing differs, because the recipe does not mention permissions. Chapter 5 adds them.

Chapter 4. Just Enough Ruby

A recipe is a Ruby program. Most of the time it reads like a list of resources, and you need no Ruby at all. But the moment you want a file per user, a package name that depends on the distribution, or the same block for three sites, Ruby is there, with nothing new to learn beyond Ruby itself. This chapter covers the part of Ruby this book uses. If you know Ruby, skim it; if you know another programming language, it will feel familiar quickly.

4.1. A recipe is a program

When sadame reads a recipe, it runs it as Ruby, top to bottom. Each resource line is a method call: `file "/tmp/hello.txt", content: "…"` calls the method `file` with a string and some options, and that call adds a resource to sadame's list. When the program ends, sadame plans or applies the list, in the order the resources were added.

So any Ruby that runs before or around those calls decides **which** resources the recipe declares. Here is a recipe that uses each building block of this chapter:

code/04-ruby/recipe.rb

```
# Just enough Ruby: plain Ruby code around file resources.

# A variable, and string interpolation with #{...}.
greeting = "Hello"

# An array, and a loop that declares one file per name.
names = ["alice", "bob"]
names.each do |name|
  file "/tmp/greeting-#{name}.txt", content: "#{greeting}, #{name}!\n"
end

# A hash: keys and values, read with settings[:key].
settings = { color: "blue", size: 3 }
file "/tmp/settings.txt", content: "color = #{settings[:color]}\nsize = #{settings[:size]}\n"

# A condition: this file only exists on Debian and Ubuntu.
if platform.family == "debian"
  file "/tmp/family.txt", content: "This machine is in the Debian family.\n"
end

# A method of your own, called like any other.
def banner(text)
  "*** #{text} ***\n"
end

file "/tmp/banner.txt", content: banner("managed by sadame")
```

And here is what it declares:

```
$ sadame plan recipe.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

+ file[/tmp/greeting-alice.txt]
    state: absent -> present
    --- /dev/null
    +++ /tmp/greeting-alice.txt (desired)
    @@ -0,0 +1 @@
    +Hello, alice!

+ file[/tmp/greeting-bob.txt]
    state: absent -> present
    --- /dev/null
```

```
    +++ /tmp/greeting-bob.txt (desired)
    @@ -0,0 +1 @@
    +Hello, bob!

+ file[/tmp/settings.txt]
    state: absent -> present
    --- /dev/null
    +++ /tmp/settings.txt (desired)
    @@ -0,0 +1,2 @@
    +color = blue
    +size = 3

+ file[/tmp/family.txt]
    state: absent -> present
    --- /dev/null
    +++ /tmp/family.txt (desired)
    @@ -0,0 +1 @@
    +This machine is in the Debian family.

+ file[/tmp/banner.txt]
    state: absent -> present
    --- /dev/null
    +++ /tmp/banner.txt (desired)
    @@ -0,0 +1 @@
    +*** managed by sadame ***

Plan: 5 to create, 0 to update, 0 to remove, 0 to run, 0 unchanged.
```

Five files, from a few lines of Ruby. The rest of this chapter takes the recipe apart.

4.2. Strings

Text in double quotes is a **string**: `"Hello"`. Inside double quotes, `\n` is a newline, and `#{…}` inserts the value of any Ruby expression, which is called **interpolation**:

```
greeting = "Hello"
"#{greeting}, alice!\n"     # => "Hello, alice!\n"
```

Single quotes, `'Hello\n'`, keep the text exactly as written: no interpolation and no `\n` newline. In recipes you will mostly want double quotes.



File content almost always ends with a newline. If you forget the `\n`, `plan` shows `\ No newline at end of file` in the diff, which is a good way to notice.

4.3. Variables

`greeting = "Hello"` stores a value under a name. Variables in a recipe are ordinary Ruby local variables: they exist from the line that sets them to the end of the file. Use them to avoid repeating a path or a name:

```
root = "/var/www/site"
directory root
file "#{root}/index.html", content: "<h1>Hi</h1>\n"
```

4.4. Symbols

Words with a colon in front, `:running`, `:absent`, `:web`, are **symbols**: names, cheaper than strings, used for fixed choices. You already met one: `state: :absent`. Resource options use symbols for

their choices (`state: :running`), and sadame uses them for group names (`:web`).

4.5. Arrays and loops

Square brackets make an **array**, a list of values: `["alice", "bob"]`. `each` runs a block of code once per element, with the element in the variable between the bars:

```
names = ["alice", "bob"]
names.each do |name|
  file "/tmp/greeting-#{name}.txt", content: "#{greeting}, #{name}!\n"
end
```

The loop runs twice, so the recipe declares two files. When you need a file per user, a directory per site, a package per entry of a list, this is how.

4.6. Hashes

Curly braces make a **hash**: pairs of keys and values. With symbol keys, Ruby lets you write `color: "blue"` for the pair:

```
settings = { color: "blue", size: 3 }
settings[:color]    # => "blue"
settings[:size]     # => 3
settings[:missing]  # => nil
```

Hashes are everywhere in sadame. The options of a resource are a hash, which is why they look like `content: "…", mode: "0644"`. The variables of chapter 6, `vars`, are a hash. Reading a key that is not there gives `nil`, Ruby's "nothing".

4.7. Conditions

`if` runs code only when a condition is true:

```
if platform.family == "debian"
  file "/tmp/family.txt", content: "This machine is in the Debian family.\n"
end
```

`platform` tells the recipe about the machine it runs on; chapter 12 covers it. `==` compares, `!=` is its opposite, and `unless` is the opposite of `if`. For a short choice between two values, Ruby has the one-line form `condition ? this : that`:

```
package platform.family == "rhel" ? "sqlite" : "sqlite3"
```

4.8. Methods

`def` defines a method of your own. Use one when the same few lines appear more than once:

```
def banner(text)
  "*** #{text} ***\n"
end

file "/tmp/banner.txt", content: banner("managed by sadame")
```

The method returns the value of its last line. Once you reach for methods often, it is time to split the recipe into files, which chapter 13 shows.

4.9. Ruby runs during plan too

Everything in a recipe runs every time sadame reads it, for `plan` and `drift` as much as for `apply`. That is fine for deciding **what** to declare. It is not fine for **changing** anything: a `File.write` at the top of a recipe would write during every `plan`. Make every change through a resource, and let plan, apply and drift do their job.



Sadame's Ruby is **mruby**, a compact Ruby built into the sadame binary. It has the core of the language (strings, arrays, hashes, blocks, methods, `JSON.parse`) but no gems and no `require`. Everything in this chapter, and in this book, works with it.

What you learned

- A recipe is a Ruby program; each resource line is a method call that adds a resource to a list.
- Strings interpolate with `#{…}`; symbols name fixed choices.
- Arrays and `each` declare a resource per element; hashes hold options and vars.
- `if` and `? :` choose; `def` defines a helper method.
- Ruby runs during `plan`, so changes belong in resources only.

Try it

1. Add `"carol"` to the `names` array and run `plan`. One more file.
2. Write a loop over `{ "red" ⇒ 1, "green" ⇒ 2 }.each do |color, number|` that declares `/tmp/color-#{color}.txt` with the number as content.
3. Change `if platform.family == "debian"` to `"rhel"` and plan again. The file disappears from the plan, because the condition is false here.

Part II: The Building Blocks

Chapter 5. Packages and Files

Most of what makes a server **your** server is software and configuration: packages installed, files with the right content and permissions. This chapter covers the two resources you will write most, `package` and `file`, and the habit of checking a file before it goes live.

From here on, recipes change the system, so they run with `sudo`:

```
$ cd ~/code/05-packages-and-files
```

5.1. Packages

code/05-packages-and-files/packages.rb

```
# Packages: installed, or absent. sadame picks apt, dnf, apk or pacman.
package "tree"
package "jq"
package "cowsay"
```

A `package` resource names a package and, by default, wants it installed. Sadame reads `/etc/os-release` to find out which package manager the machine uses, `apt` on Debian and Ubuntu, `dnf` on Rocky Linux, `apk` on Alpine, `pacman` on Arch, and asks it whether the package is there. The plan:

```
$ sudo sadame plan packages.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

+ package[tree]
    state: absent -> installed

+ package[jq]
    state: absent -> installed

+ package[cowsay]
    state: absent -> installed

Plan: 3 to create, 0 to update, 0 to remove, 0 to run, 0 unchanged.
```

Applying it installs the three packages; then cowsay can say hello:

```
$ sudo sadame apply packages.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

changed    package[tree]
           state: absent -> installed
changed    package[jq]
           state: absent -> installed
changed    package[cowsay]
           state: absent -> installed

Summary
  changed      3  package[tree]
                  package[jq]
                  package[cowsay]
  unchanged    0
  failed       0
```

```
  skipped      0
  notified     0

Finished in 4.96s.

$ /usr/games/cowsay 'Hello from sadame'
 ___________________
< Hello from sadame >
 -------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

To remove a package, declare it absent:

code/05-packages-and-files/no-cowsay.rb

```
# Removing a package is declaring it absent.
package "cowsay", state: :absent
```

```
$ sudo sadame apply no-cowsay.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

changed    package[cowsay]
           state: installed -> absent

Summary
  changed      1  package[cowsay]
  unchanged    0
  failed       0
  skipped      0
  notified     0

Finished in 0.41s.
```

That is nearly all there is to `package`. Its only other option, `options:`, passes extra flags to the package manager, such as `options: "--no-install-recommends"`.



`package` makes sure a package is installed; it does not upgrade it to the newest version. Keeping packages up to date is the job of your distribution's automatic updates, which chapter 17 switches on with a recipe.

5.2. Files with content

You met the `file` resource in chapter 3. Here it does more:

code/05-packages-and-files/files.rb

```
# Files: content written in the recipe, copied from files/, or removed.

# Content in the recipe, with permissions and an owner.
file "/etc/motd",
     content: "This machine is managed by sadame. Changes made by hand will be undone.\n",
     mode: "0644",
     owner: "root",
     group: "root"

# A script copied from files/hello.sh, made executable.
file "/usr/local/bin/hello", source: "hello.sh", mode: "0755"

# A file that must not exist.
```

```
file "/etc/motd.old", state: :absent
```

Three resources, three ways to say what a file holds:

`content:`

: The text, written in the recipe. Best for short files.

`source:`

: A file from the recipe's directory, or from its `files/` subdirectory, copied as it is. Best for scripts and longer files.

`state: :absent`

: No file at all.

`mode:`, `owner:` and `group:` set the permissions and ownership. Write the mode as a string of octal digits, `"0644"` or `"0755"`, as `chmod` takes it. Options you leave out are left alone: a recipe without `mode:` accepts whatever mode the file has.

The `files/hello.sh` the second resource copies:

code/05-packages-and-files/files/hello.sh

```
#!/bin/sh
# Installed by sadame from files/hello.sh.
echo "Hello, $(whoami), from $(hostname)!"
```

Apply the recipe and run the new command:

```
$ sudo sadame apply files.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

changed    file[/etc/motd]
           --- /etc/motd
           +++ /etc/motd (desired)
           @@ -1,7 +1 @@
           -
           -The programs included with the Debian GNU/Linux system are free software;
           -the exact distribution terms for each program are described in the
           -individual files in /usr/share/doc/*/copyright.
           -
           -Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
           -permitted by applicable law.
           +This machine is managed by sadame. Changes made by hand will be undone.
changed    file[/usr/local/bin/hello]
           state: absent -> present
           mode: 0755
           --- /dev/null
           +++ /usr/local/bin/hello (desired)
           @@ -0,0 +1,3 @@
           +#!/bin/sh
           +# Installed by sadame from files/hello.sh.
           +echo "Hello, $(whoami), from $(hostname)!"
unchanged  file[/etc/motd.old]

Summary
  changed      2  file[/etc/motd]
                  file[/usr/local/bin/hello]
  unchanged    1
  failed       0
  skipped      0
  notified     0

Finished in 0.01s.
```

```
$ hello
Hello, ops, from practice!
```

The diff for `/etc/motd` shows Debian's original message being replaced, line by line. `/etc/motd.old` is `unchanged`: it does not exist, which is what the recipe asks for.

5.3. Permissions drift too

Content is not the only thing that drifts. Change the script's mode by hand and plan again:

```
$ sudo chmod 0700 /usr/local/bin/hello

$ sudo sadame plan files.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

~ file[/usr/local/bin/hello]
    mode: 0700 -> 0755

Plan: 0 to create, 1 to update, 0 to remove, 0 to run, 2 unchanged.
```

The plan shows the one attribute that differs, `mode`, as `from → to`. Applying it puts `0755` back, without rewriting the content.

5.4. How sadame writes a file

When the content changes, sadame does not overwrite the file in place. It writes the new content to a temporary file next to it, gives that file its final mode and owner, and then renames it over the original. A program reading the file at that moment sees either the old version or the new one, never half of each. That matters for configuration that services reload while you change it.

5.5. Check before you replace

Some files can lock you out when they are wrong. A broken file in `/etc/sudoers.d` makes `sudo` refuse to work for everyone, and on a server reached over SSH that can mean losing your only way in. For such files the `verify:` option runs a command on the **temporary** file before it replaces the real one; `%{path}` in the command is replaced by its path:

code/05-packages-and-files/sudoers.rb

```
# A sudoers drop-in that is checked with visudo before it replaces anything.
# A broken sudoers file can lock everyone out of sudo; verify: makes sadame
# refuse to install one.
file "/etc/sudoers.d/ops-tools",
     content: "ops ALL=(root) NOPASSWD: /usr/bin/apt-get update\n",
     mode: "0440",
     verify: "visudo -cf %{path}"
```

`visudo -cf FILE` checks a sudoers file without installing it. With a correct file, apply works as usual:

```
$ sudo sadame apply sudoers.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))
```

```
changed    file[/etc/sudoers.d/ops-tools]
           state: absent -> present
           mode: 0440
           --- /dev/null
           +++ /etc/sudoers.d/ops-tools (desired)
           @@ -0,0 +1 @@
           +ops ALL=(root) NOPASSWD: /usr/bin/apt-get update

Summary
  changed      1  file[/etc/sudoers.d/ops-tools]
  unchanged    0
  failed       0
  skipped      0
  notified     0

Finished in 0.00s.
```

Now the same file with a typo, `NOPASWD` instead of `NOPASSWD`:

```
$ sudo sadame apply sudoers-typo.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

failed     file[/etc/sudoers.d/ops-tools]
           --- /etc/sudoers.d/ops-tools
           +++ /etc/sudoers.d/ops-tools (desired)
           @@ -1 +1 @@
           -ops ALL=(root) NOPASSWD: /usr/bin/apt-get update
           +ops ALL=(root) NOPASWD: /usr/bin/apt-get update
           error: file[/etc/sudoers.d/ops-tools]: verify failed, /etc/sudoers.d/ops-tools was not changed:
visudo -cf /etc/sudoers.d/ops-tools.sadame-new
           /etc/sudoers.d/ops-tools.sadame-new:1:48: syntax error
           ops ALL=(root) NOPASWD: /usr/bin/apt-get update
                                                   ^
           /etc/sudoers.d/ops-tools.sadame-new:1:48: syntax error
           ops ALL=(root) NOPASWD: /usr/bin/apt-get update
                                                   ^

Summary
  changed      0
  unchanged    0
  failed       1  file[/etc/sudoers.d/ops-tools]
  skipped      0
  notified     0

Finished in 0.00s.

$ echo $?
1

$ sudo cat /etc/sudoers.d/ops-tools
ops ALL=(root) NOPASSWD: /usr/bin/apt-get update
```

`visudo` rejected the new content, so sadame left the real file alone, removed the temporary one, reported the resource as `failed`, and exited with `1`. The last command shows the file is still the good version.



Many programs can check a configuration file without loading it: `nginx -t -c FILE`, `sshd -t -f FILE`, `visudo -cf FILE`, `named-checkconf FILE`. Whenever a file has such a checker, use `verify:`.

What you learned

- `package` installs or removes a package with the machine's own package manager.
- `file` takes its content from `content:` or `source:`, or is absent; `mode:`, `owner:` and `group:` set

permissions.

- Attributes you do not mention are left alone; those you do are checked for drift.
- Files are replaced in one rename, never half-written.
- `verify:` checks a file before it goes live, and a failed check changes nothing.

Try it

1. Add `package "htop"` to `packages.rb`, plan and apply.
2. Give `/etc/motd` a second line, then edit the file by hand and run `sudo sadame drift files.rb`. Which lines does the diff mark?
3. Change the `hello.sh` script in `files/` and plan. The diff is the change you made.

Chapter 6. Templates and Variables

Real configuration files are mostly the same from machine to machine, with a few values that differ: a port, a number of workers, a list of features. Copying a whole file per machine would bury those few values. Instead, sadame keeps them as **vars** and fills them into **templates**.

```
$ cd ~/code/06-templates-and-vars
```

6.1. vars.rb: the values in one place

Next to a recipe, a file called `vars.rb` holds its default values. It is Ruby too: a hash, the last thing in the file.

code/06-templates-and-vars/vars.rb

```
# Default values for the recipe and its templates, read as vars[:name].
{
  app_name: "inventory",
  port: 8080,
  workers: 2,
  features: ["search", "export"],
  debug: false
}
```

Sadame loads it before the recipe, and the recipe and its templates read it as `vars`: `vars[:port]` is `8080`, `vars[:features]` is the array of two strings. Values can be any Ruby value: strings, numbers, booleans, arrays, hashes.

6.2. Templates

A template is a text file with gaps that Ruby fills in. Sadame's templates use **ERB**, the format Ruby on Rails and many other tools use:

code/06-templates-and-vars/templates/app.conf.erb

```
# <%= vars[:app_name] %> configuration. Managed by sadame.
[server]
port = <%= vars[:port] %>
workers = <%= vars[:workers] %>
<% if vars[:debug] -%>
log_level = debug
<% end -%>

[features]
<% vars[:features].each do |feature| -%>
<%= feature %> = enabled
<% end -%>
```

Three kinds of tags do all the work:

Tag	Effect
`<%= expression %>`	Inserts the value of the expression: `<%= vars[:port] %>` becomes `8080`.
`<% code %>`	Runs Ruby without inserting anything: the `if` and the `each` loop above.

Tag	Effect
`-%>`	At the end of a tag: also removes the line break after it, so lines holding only code leave no blank lines behind.

The recipe asks for the template with the `template:` option instead of `content:`:

code/06-templates-and-vars/recipe.rb

```
# A configuration file rendered from an ERB template with the vars.
directory "/etc/inventory", mode: "0755"

file "/etc/inventory/app.conf",
     template: "app.conf.erb",
     mode: "0644"

# locals: hands values to one template only.
file "/etc/inventory/maintainer.txt",
     template: "maintainer.txt.erb",
     locals: { team: "platform", email: "platform@example.com" },
     mode: "0644"
```

Sadame looks for `app.conf.erb` next to the recipe first, then in its `templates/` directory. The plan shows the file as it will be written, with the template already filled in:

```
$ sudo sadame plan recipe.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

+ directory[/etc/inventory]
    state: absent -> present
    mode: 0755

+ file[/etc/inventory/app.conf]
    state: absent -> present
    mode: 0644
    --- /dev/null
    +++ /etc/inventory/app.conf (desired)
    @@ -0,0 +1,8 @@
    +# inventory configuration. Managed by sadame.
    +[server]
    +port = 8080
    +workers = 2
    +
    +[features]
    +search = enabled
    +export = enabled

+ file[/etc/inventory/maintainer.txt]
    state: absent -> present
    mode: 0644
    --- /dev/null
    +++ /etc/inventory/maintainer.txt (desired)
    @@ -0,0 +1 @@
    +Maintained by the platform team: platform@example.com

Plan: 3 to create, 0 to update, 0 to remove, 0 to run, 0 unchanged.
```

`debug` is `false`, so there is no `log_level` line; the loop wrote one line per feature.

6.3. locals: values for one template

`vars` are shared by the whole recipe. When a value belongs to one file only, hand it to that template with `locals:`, and read it there as `locals[:name]`:

code/06-templates-and-vars/templates/maintainer.txt.erb

```
Maintained by the <%= locals[:team] %> team: <%= locals[:email] %>
```

`locals` are also how you pass values a template cannot see by itself, such as the host's name in chapter 11. A template sees `vars`, `locals`, and the machine's `facts` (chapter 12), and nothing else from the recipe.

6.4. Changing a value from the command line

After applying the recipe, try different values without editing anything: `--var name=value` overrides a var for one run.

```
$ sudo sadame plan recipe.rb --var port=9090 --var debug=true
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

~ file[/etc/inventory/app.conf]
    --- /etc/inventory/app.conf
    +++ /etc/inventory/app.conf (desired)
    @@ -1,7 +1,8 @@
     # inventory configuration. Managed by sadame.
     [server]
    -port = 8080
    +port = 9090
     workers = 2
    +log_level = debug

     [features]
     search = enabled

Plan: 0 to create, 1 to update, 0 to remove, 0 to run, 2 unchanged.
```

The diff shows exactly what those two values would change in the rendered file: the port, and a new `log_level` line. On the command line, whole numbers become numbers and `true` or `false` become booleans; everything else stays a string.

This is one of the most useful habits sadame enables: **try a change with `plan` and `--var` first**, read the diff, and only then put the value in `vars.rb` and apply.

6.5. Where vars come from

So far vars came from two places. Chapter 11 adds two more, the inventory's groups and hosts. When the same var is set in several places, the most specific wins:

1. `vars.rb` next to the recipe: the defaults.
2. Vars of the host's groups, in `hosts.rb` (chapter 11).
3. Vars of the host itself, in `hosts.rb`.
4. `--var` on the command line: always wins.

6.6. When a template is wrong

A mistake in a template, such as a method that does not exist or arithmetic on a var nobody set, is an error in Ruby code like any other: the resource fails, and sadame names the template's file

and line, as chapter 20 shows. A misspelled var, `vars[:prot]`, is quieter: it is `nil`, which prints as nothing. Where a var must be set, write `vars.fetch(:port)`, which fails loudly when it is not. You can catch such mistakes before they reach a machine with `sadame validate` (chapter 19).

What you learned

- `vars.rb` holds the recipe's default values as a Ruby hash; recipes and templates read them as `vars[:name]`.
- `file` with `template:` renders an ERB template from `templates/`.
- `<%= %>` inserts a value, `<% %>` runs code, `-%>` swallows the line break.
- `locals:` gives one template its own values.
- `--var` overrides a var for one run, which makes `plan` a safe way to try values.

Try it

1. Add `"reports"` to `features` in `vars.rb` and plan. One more line in the diff.
2. Add a var `timeout: 30` and a `timeout = <%= vars[:timeout] %>` line in the template under `[server]`. Plan, then apply.
3. Run `sudo sadame plan recipe.rb --var workers=8`. Then add `workers: 8` to `vars.rb` and plan without `--var`: the same diff.

Chapter 7. Directories, Links and Lines

Three smaller resources round out the file system: `directory` for folders, `link` for symbolic links, and `line` for the case where a file belongs to someone else and you only need one line of it. This chapter builds a tiny site with two releases side by side, a link that says which one is live, and changes one setting in a file Debian owns.

```
$ cd ~/code/07-directories-links-lines
```

code/07-directories-links-lines/recipe.rb

```
# Two releases of a small site side by side, a link that says which one is
# live, and one line changed in a file a package owns.

[1, 2].each do |number|
  directory "/srv/site/releases/#{number}", mode: "0755"
  file "/srv/site/releases/#{number}/index.html",
       content: "<h1>Release #{number}</h1>\n",
       mode: "0644"
end

# The live release: switching is one change to this link.
link "/srv/site/current", to: "releases/#{vars[:release]}"

# /etc/login.defs comes with Debian. Only one setting is ours: passwords
# of new accounts expire after a year.
line "PASS_MAX_DAYS 365", path: "/etc/login.defs", match: /^PASS_MAX_DAYS\s/

# A directory that must not exist, with everything in it.
directory "/srv/site/tmp", state: :absent, recursive: true
```

`vars.rb` sets `release: 2`. Apply it and look at the result:

```
$ sudo sadame apply recipe.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

changed    directory[/srv/site/releases/1]
           state: absent -> present
           mode: 0755
changed    file[/srv/site/releases/1/index.html]
           state: absent -> present
           mode: 0644
           --- /dev/null
           +++ /srv/site/releases/1/index.html (desired)
           @@ -0,0 +1 @@
           +<h1>Release 1</h1>
changed    directory[/srv/site/releases/2]
           state: absent -> present
           mode: 0755
changed    file[/srv/site/releases/2/index.html]
           state: absent -> present
           mode: 0644
           --- /dev/null
           +++ /srv/site/releases/2/index.html (desired)
           @@ -0,0 +1 @@
           +<h1>Release 2</h1>
changed    link[/srv/site/current]
           state: absent -> present
           to: releases/2
changed    line[PASS_MAX_DAYS 365]
           --- /etc/login.defs
           +++ /etc/login.defs (desired)
           @@ -162,7 +162,7 @@
            #   PASS_MIN_DAYS   Minimum number of days allowed between password changes.
            #   PASS_WARN_AGE   Number of days warning given before a password expires.
            #
           -PASS_MAX_DAYS   99999
```

```
            +PASS_MAX_DAYS 365
             PASS_MIN_DAYS   0
             PASS_WARN_AGE   7

unchanged  directory[/srv/site/tmp]

Summary
  changed      6  directory[/srv/site/releases/1]
                  file[/srv/site/releases/1/index.html]
                  directory[/srv/site/releases/2]
                  file[/srv/site/releases/2/index.html]
                  link[/srv/site/current]
                  line[PASS_MAX_DAYS 365]
  unchanged    1
  failed       0
  skipped      0
  notified     0

Finished in 0.01s.

$ readlink /srv/site/current
releases/2

$ grep ^UMASK /etc/login.defs
UMASK       022
```

7.1. Directories

`directory` makes sure a directory exists, with the `mode:`, `owner:` and `group:` you give, or does not exist. Missing parent directories are created along the way: declaring `/srv/site/releases/1` also creates `/srv/site` and `/srv/site/releases` if they are missing.

To remove a directory, declare it `state: :absent`. Sadame removes only empty directories, unless you add `recursive: true`, as the recipe does for `/srv/site/tmp`. That is deliberate: deleting a tree should be something you asked for in writing.



Parents created along the way get default permissions. When their mode or owner matters, declare them as resources of their own.

7.2. Links

`link` manages a **symbolic link**: a small file that points to another path. The site keeps every release in its own directory and points `/srv/site/current` at the live one. Anything that serves the site reads from `current`, and switching releases is one change to the link.

The target is stored as written. `to: "releases/2"` is a **relative** target, resolved from the link's own directory, which keeps the whole tree movable. Try switching to release 1 with a var:

```
$ sudo sadame plan recipe.rb --var release=1
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

~ link[/srv/site/current]
    to: releases/2 -> releases/1

Plan: 0 to create, 1 to update, 0 to remove, 0 to run, 6 unchanged.
```

One change: the link's target, `to: releases/2 → releases/1`.

If a regular file is where the link should be, sadame refuses to replace it, unless you add `force:`

`true`; a directory in the way is always an error. Typical uses of `link` are the `sites-enabled` links of nginx, which you will meet in chapter 14, the time zone in `/etc/localtime`, and release switching as here.

7.3. Lines

Some files belong to a package: `/etc/login.defs` here, `postgresql.conf` and `pg_hba.conf` in chapter 15. Taking over the whole file with a `file` resource would mean copying the distribution's version into your recipe and keeping it in step with every update. Usually you need one setting. That is what `line` is for.

```
line "PASS_MAX_DAYS 365", path: "/etc/login.defs", match: /^PASS_MAX_DAYS\s/
```

The name is the line you want. `path:` is the file. `match:` is a **regular expression**, a pattern for the lines this one replaces: `/^PASS_MAX_DAYS\s/` means "lines that start with `PASS_MAX_DAYS` followed by a space or tab". The rule is simple:

- every line that matches `match` is replaced by your line;
- when none matches and your line is not in the file yet, it is appended at the end;
- with `state: :absent`, your line, and every line that matches, is removed.

The diff in the apply output above shows exactly that: Debian's `PASS_MAX_DAYS 99999`, which means "never", became `PASS_MAX_DAYS 365`, in its place, with the rest of the file untouched. (The setting applies to accounts created from now on: passwords expire after a year.)



Write `match` so that it covers every form the setting can take in the file: commented out or not, with spaces or tabs. `/^#?\s*include_dir\s*=/` matches both `include_dir = 'conf.d'` and the commented-out `#include_dir = '…'` of a stock PostgreSQL configuration. Chapter 15 uses exactly this.

`line` keeps the file's mode and owner, writes it in one rename like `file` does, and takes `verify:` too. It edits existing files only: if the file does not exist when the change is applied, it fails rather than create a file with one line in it.

7.4. Which resource for a file?

You want	Use
A file that is entirely yours	`file` with `content:`, `source:` or `template:`
One setting in a file a package owns	`line`
A file in a directory that may not exist yet	`directory`, then `file`
A second name for a file or directory	`link`

Never manage the same file with both `file` and `line`: the `file` resource writes its content on every run and `line` changes it again, and neither ever reports `unchanged`.

What you learned

- `directory` creates missing parents; removing a non-empty directory needs `recursive: true`.
- `link` points a path at a target, stored as written; relative targets resolve from the link's directory.
- `line` changes one line in a file someone else owns: matching lines are replaced, otherwise the line is appended.
- Choose `file` for whole files, `line` for single settings, never both.

Try it

1. Add a third release to the loop, `[1, 2, 3]`, and switch to it with `--var release=3`.
2. Make `/srv/site/tmp` exist with `sudo mkdir -p /srv/site/tmp/cache`, then plan. The recipe removes it, cache and all.
3. Add `line "PASS_MIN_DAYS 1", path: "/etc/login.defs", match: /^PASS_MIN_DAYS\s/` and plan. Read the diff before you apply it.

Chapter 8. Services and Notifications

Installing a program is half the job; the other half is keeping it running, starting it at boot, and reloading it when its configuration changes. This chapter installs nginx on the practice machine, keeps it running with the `service` resource, and connects the two with a **notification**: when the configuration file changes, nginx reloads.

```
$ cd ~/code/08-services-and-notifications
```

code/08-services-and-notifications/recipe.rb

```
# nginx installed, running, started at boot, and reloaded whenever its
# configuration changes.
package "nginx"

file "/etc/nginx/conf.d/status.conf",
     template: "status.conf.erb",
     mode: "0644",
     notifies: [:reload, "service[nginx]"]

service "nginx", enable: true, state: :running
```

The template is an nginx server block for nginx's own status page, on the port `vars.rb` sets (`status_port: 8080`):

code/08-services-and-notifications/templates/status.conf.erb

```
# nginx's own status page. Managed by sadame.
server {
    listen 127.0.0.1:<%= vars[:status_port] %>;
    location = /status {
        stub_status;
    }
}
```

8.1. The service resource

`service` manages two things about a service, each only when you give the option:

`enable: true` **or** `false`

: Whether it starts at boot. With systemd, this is `systemctl enable` and `disable`.

`state: :running` **or** `:stopped`

: Whether it runs right now: `systemctl start` and `stop`.

Sadame reads the current state with `systemctl is-enabled` and `systemctl is-active` and changes only what differs. On Alpine, which uses OpenRC instead of systemd, the same recipe works with OpenRC's commands.

8.2. Notifications

The `file` resource has one option you have not seen:

```
notifies: [:reload, "service[nginx]"]
```

It means: **when this file changes, reload** `service[nginx]`. If the file is already right, nothing happens. The plan shows the notification under the resource that would trigger it:

```
$ sudo sadame plan recipe.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

+ package[nginx]
    state: absent -> installed

+ file[/etc/nginx/conf.d/status.conf]
    state: absent -> present
    mode: 0644
    --- /dev/null
    +++ /etc/nginx/conf.d/status.conf (desired)
    @@ -0,0 +1,7 @@
    +# nginx's own status page. Managed by sadame.
    +server {
    +    listen 127.0.0.1:8080;
    +    location = /status {
    +        stub_status;
    +    }
    +}
    notifies: reload service[nginx] (delayed)

~ service[nginx]
    enabled: false -> true
    state: stopped -> running

Plan: 2 to create, 1 to update, 0 to remove, 0 to run, 0 unchanged.
```

Now apply it:

```
$ sudo sadame apply recipe.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

changed    package[nginx]
           state: absent -> installed
changed    file[/etc/nginx/conf.d/status.conf]
           state: absent -> present
           mode: 0644
           --- /dev/null
           +++ /etc/nginx/conf.d/status.conf (desired)
           @@ -0,0 +1,7 @@
           +# nginx's own status page. Managed by sadame.
           +server {
           +    listen 127.0.0.1:8080;
           +    location = /status {
           +        stub_status;
           +    }
           +}
changed    service[nginx]
           state: stopped -> running
notified   service[nginx]: reload (delayed, from file[/etc/nginx/conf.d/status.conf])

Summary
  changed      3  package[nginx]
                  file[/etc/nginx/conf.d/status.conf]
                  service[nginx]
  unchanged    0
  failed       0
  skipped      0
  notified     1  service[nginx]: reload

Finished in 2.12s.

$ curl -s http://127.0.0.1:8080/status
Active connections: 1
server accepts handled requests
 1 1 1
Reading: 0 Writing: 1 Waiting: 0
```

Look closely at `service[nginx]`. The plan said `enabled: false → true` and `state: stopped → running`, but the apply only changed `state`. Installing the Debian package enabled the service on its own; by the time sadame reached `service[nginx]`, there was less to do.

That is how apply works: resources run in the order of the recipe, and each one reads the machine's state **right before** it acts. A plan is a prediction from the machine as it is now; apply adapts as earlier resources change things.

The notification ran last, after every resource, as `notified`. That is the default timing, `delayed`: if three files notify nginx, it reloads once, at the end, when all three are written. The status page answers.

8.3. A change triggers the reload

Move the status page to another port, for this run, with a var:

```
$ sudo sadame apply recipe.rb --var status_port=8081
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

unchanged  package[nginx]
changed    file[/etc/nginx/conf.d/status.conf]
           --- /etc/nginx/conf.d/status.conf
           +++ /etc/nginx/conf.d/status.conf (desired)
           @@ -1,6 +1,6 @@
            # nginx's own status page. Managed by sadame.
            server {
           -    listen 127.0.0.1:8080;
           +    listen 127.0.0.1:8081;
                location = /status {
                    stub_status;
                }
unchanged  service[nginx]
notified   service[nginx]: reload (delayed, from file[/etc/nginx/conf.d/status.conf])

Summary
  changed      1  file[/etc/nginx/conf.d/status.conf]
  unchanged    2
  failed       0
  skipped      0
  notified     1  service[nginx]: reload

Finished in 0.01s.

$ curl -s --retry 5 --retry-connrefused http://127.0.0.1:8081/status
Active connections: 1
server accepts handled requests
 2 2 2
Reading: 0 Writing: 1 Waiting: 0
```

The file changed, so the reload ran, and nginx answers on the new port. The service itself was `unchanged`: it was enabled and running already. The reload came from the notification, not from the service's own options.

8.4. Services drift too

Stop nginx by hand and ask for drift:

```
$ sudo systemctl stop nginx

$ sudo sadame drift recipe.rb --var status_port=8081
```

```
sadame 0.5.1 drift on local (Debian GNU/Linux 12 (bookworm))

~ service[nginx]
    state: stopped -> running

Drift: 1 of 3 resources differs from the recipe.

$ echo $?
1
```

The service should run and does not: drift, exit status `1`. The next `apply` starts it again.

8.5. Actions and timing

`service` accepts six actions from notifications: `:start`, `:stop`, `:restart`, `:reload`, `:enable` and `:disable`. Use `:reload` when the program can reread its configuration without dropping connections, as nginx can, and `:restart` otherwise.

Add `:immediately` to run the action right after the resource that changed, instead of at the end:

```
notifies: [:restart, "service[app]", :immediately]
```

You need that when a later resource depends on the action having happened; chapter 16 uses it to reload systemd's unit files before the service that needs them starts. A resource can notify several others with a list: `notifies: [[:reload, "service[nginx]"], [:run, "exec[…]"]]`.

The other direction is `subscribes:`, declared on the resource that reacts: `service "nginx", subscribes: [:reload, "file[/etc/nginx/nginx.conf]"]`. Both mean the same; use whichever reads better where you write it.



A notification to a resource that is not declared is an error, reported before anything runs. A typo in `"service[ngnix]"` cannot slip through; chapter 20 shows the message.

What you learned

- `service` manages `enable:` (at boot) and `state:` (now), each only when given.
- `notifies:` runs an action on another resource when this one changes; delayed notifications run once each, at the end.
- `:immediately` runs the action right away; `subscribes:` declares the same link from the other side.
- Apply reads each resource's state right before acting, so it adapts to what earlier resources did.

Try it

1. Change `stub_status` to a `return 200 "ok\n";` line in the template and apply. Watch for the `notified` line, then `curl` the page.
2. Remove `notifies:` from the recipe, change the port again and apply. Does nginx answer on

the new port? Why not?

3. Add `service "cron", enable: true, state: :running` and plan. On a machine where cron already runs, it is unchanged.

Chapter 9. Users, Groups and Commands

A server is used by people and by programs, each with an account and the right groups. And sometimes no resource fits what you need to do, and you need to run a command. This chapter covers `user`, `group`, and `exec`, the resource for commands, with the guards that keep commands as safe to repeat as everything else.

```
$ cd ~/code/09-users-groups-commands
```

code/09-users-groups-commands/recipe.rb

```
# A team's accounts, a shared directory, and commands that run only when
# they have something to do.

group "developers", gid: 2000

user "alice", shell: "/bin/bash", groups: ["developers"]
user "bob", shell: "/bin/bash", groups: ["developers"]

# Owned by the group; mode 2775 keeps new files in the group too.
directory "/srv/shared", owner: "root", group: "developers", mode: "2775"

# Runs once: creates: names the file the command makes.
exec "ssh-keygen -q -t ed25519 -N '' -C deploy -f /srv/shared/deploy_key",
     creates: "/srv/shared/deploy_key"

# Runs when the guard says the work is not done yet. The command needs git,
# so the recipe says so.
package "git"
exec "git config --system init.defaultBranch main",
     not_if: "git config --system --get init.defaultBranch | grep -qx main"

# Runs only when notified: refresh apt's package lists after its
# configuration changes.
file "/etc/apt/apt.conf.d/99no-recommends",
     content: "APT::Install-Recommends \"false\";\n",
     mode: "0644",
     notifies: [:run, "exec[apt-get update]"]

exec "apt-get update", action: :nothing
```

The plan:

```
$ sudo sadame plan recipe.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

+ group[developers]
    state: absent -> present
    gid: 2000

+ user[alice]
    state: absent -> present
    shell: /bin/bash
    groups: developers

+ user[bob]
    state: absent -> present
    shell: /bin/bash
    groups: developers

+ directory[/srv/shared]
    state: absent -> present
    mode: 2775
    owner: root
    group: developers

! exec[ssh-keygen -q -t ed25519 -N '' -C deploy -f /srv/shared/deploy_key]
```

```
    command: ssh-keygen -q -t ed25519 -N '' -C deploy -f /srv/shared/deploy_key
    (/srv/shared/deploy_key does not exist)

! exec[git config --system init.defaultBranch main]
    command: git config --system init.defaultBranch main
    (not_if exited non-zero)

+ file[/etc/apt/apt.conf.d/99no-recommends]
    state: absent -> present
    mode: 0644
    --- /dev/null
    +++ /etc/apt/apt.conf.d/99no-recommends (desired)
    @@ -0,0 +1 @@
    +APT::Install-Recommends "false";
    notifies: run exec[apt-get update] (delayed)

Plan: 5 to create, 0 to update, 0 to remove, 2 to run, 2 unchanged.
```

The `!` marks are new: those resources are commands that would **run**. More on them below. First, the accounts.

9.1. Groups and users

`group` makes sure a group exists, with the `gid:` you give, or does not exist. Giving a fixed gid keeps group ownership consistent across servers and in backups. `system: true` takes a gid from the system range, for service accounts.

`user` manages an account: `uid:`, `home:`, `shell:`, `system:`, and `groups:`, the groups the user must belong to. Two details matter:

- `groups:` is a minimum. The user must be in each group listed; other memberships are left alone. That keeps sadame from removing someone from a group another tool or person added them to.
- The groups must exist. Declare the `group` before the `user`, as the recipe does; resources run in order.

New users get a home directory unless they are system users. `state: :absent` deletes the account with `userdel` and keeps the home directory, because deleting someone's files should be its own decision.

The directory `/srv/shared` belongs to the `developers` group with mode `2775`: the `2` in front is the **setgid** bit, which makes new files in the directory belong to the group too, so the team can share them.

9.2. Commands with exec

`exec` runs a shell command. It is the escape hatch for everything no resource covers: generating a key, initializing a database, calling a vendor's setup tool. It is also the one resource that can break idempotence, because sadame cannot read the state a command produces.

So `exec` takes **guards**: options that tell sadame when the work is already done.

`creates: "PATH"`

: Run only while `PATH` does not exist. Best when the command makes a file, as `ssh-keygen` does.

`not_if: "COMMAND"`

Run only when `COMMAND` fails (exits non-zero): "not if this is already true".

`only_if: "COMMAND"`

Run only when `COMMAND` succeeds.

Notice `package "git"` right before the `git config` command. A command can only run if its program is installed, and a recipe should not count on somebody having installed it. On the practice machine git is there already, so the package is unchanged; on a fresh server, it is installed first. Testing this book's recipes on new virtual machines found exactly this omission.

The plan shows the reason a command would run, in parentheses: `(/srv/shared/deploy_key does not exist)`, `(not_if exited non-zero)`.



Guards run during `plan` and `drift` too, so they must only **read**. A guard that changes something would change it on every plan.

An `exec` without any guard runs on every `apply`, and `plan` and `drift` always list it. That is sometimes right, but usually a sign that a guard is missing.

9.3. Commands that run only when notified

The last two resources work as a pair. `exec "apt-get update"` has `action: :nothing`: it never runs on its own. The file resource above it notifies it with `:run`. So apt refreshes its package lists exactly when its configuration changes, and never otherwise:

```
$ sudo sadame apply recipe.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

changed    group[developers]
           state: absent -> present
           gid: 2000
changed    user[alice]
           state: absent -> present
           shell: /bin/bash
           groups: developers
changed    user[bob]
           state: absent -> present
           shell: /bin/bash
           groups: developers
changed    directory[/srv/shared]
           state: absent -> present
           mode: 2775
           owner: root
           group: developers
changed    exec[ssh-keygen -q -t ed25519 -N '' -C deploy -f /srv/shared/deploy_key]
           command: ssh-keygen -q -t ed25519 -N '' -C deploy -f /srv/shared/deploy_key
           (/srv/shared/deploy_key does not exist)
unchanged  package[git]
changed    exec[git config --system init.defaultBranch main]
           command: git config --system init.defaultBranch main
           (not_if exited non-zero)
changed    file[/etc/apt/apt.conf.d/99no-recommends]
           state: absent -> present
           mode: 0644
           --- /dev/null
           +++ /etc/apt/apt.conf.d/99no-recommends (desired)
           @@ -0,0 +1 @@
           +APT::Install-Recommends "false";
unchanged  exec[apt-get update]
```

```
notified   exec[apt-get update]: run (delayed, from file[/etc/apt/apt.conf.d/99no-recommends])

Summary
  changed      7  group[developers]
                  user[alice]
                  user[bob]
                  directory[/srv/shared]
                  exec[ssh-keygen -q -t ed25519 -N '' -C deploy -f /srv/shared/deploy_key]
                  exec[git config --system init.defaultBranch main]
                  file[/etc/apt/apt.conf.d/99no-recommends]
  unchanged    2
  failed       0
  skipped      0
  notified     1  exec[apt-get update]: run

Finished in 0.57s.

$ id alice
uid=1001(alice) gid=1001(alice) groups=1001(alice),2000(developers)

$ ls -ld /srv/shared
drwxrwsr-x 1 root developers 48 Nov  2 08:51 /srv/shared
```

`exec[apt-get update]` is reported `unchanged` in its own place, then `notified` at the end, when the configuration file it watches had changed. This pattern, a command with `action: :nothing` triggered by a notification, is how you attach "and then run this" to a change.

9.4. Nothing left to do

Plan once more:

```
$ sudo sadame plan recipe.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

Plan: 0 to create, 0 to update, 0 to remove, 0 to run, 9 unchanged.
```

Every guard now says the work is done: the key exists, the git setting is `main`, the apt file is unchanged so nothing notifies `apt-get update`. Nine resources, all unchanged. A recipe full of commands can be as calm on its second run as a recipe of files, as long as every command has a guard.

What you learned

- `group` and `user` manage accounts; `groups:` is a minimum, and groups must be declared before their users.
- `exec` runs a command; `creates:`, `not_if:` and `only_if:` make it run only when needed.
- Guards run during `plan` too, so they must only read.
- `action: :nothing` plus a notification runs a command only when something changed.

Try it

1. Add a user `carol` with `groups: ["developers"]`, plan and apply. Check with `id carol`.
2. Run `sudo rm /srv/shared/deploy_key` and plan. The key's `exec` is back in the plan, with its reason.
3. Write an `exec` with no guard, say `exec "date > /tmp/last-run"`, and run `sudo sadame drift`

`recipe.rb`. Why does drift now always exit with `1`?

Part III: Many Machines

Chapter 10. Reaching Servers over SSH

So far every recipe ran on the practice machine itself. Real servers are somewhere else: in a data center, at a cloud provider, under a desk. This chapter runs a recipe on the lab's servers from the practice machine, over SSH, and looks at what sadame does to make that work.

```
$ cd ~/code/10-remote-hosts
```

code/10-remote-hosts/motd.rb

```
# A login message that names the server it is on.
file "/etc/motd",
     content: "Welcome to #{host_name}. This server is managed by sadame.\n",
     mode: "0644"
```

`host_name` is the name of the machine the recipe is running for, so the same recipe writes a different message on each server.

10.1. One option makes it remote

Add `-h` and a host name, and the recipe runs there instead of here:

```
$ sadame plan motd.rb -h web1
[web1] copying sadame-linux-arm64 to
~/.sadame/bin/dcf637ae7d2f79587cd36cc4fb47eff845444424933647a5e75565917475c21d/
[web1] sadame 0.5.1 plan on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] ~ file[/etc/motd]
[web1]     --- /etc/motd
[web1]     +++ /etc/motd (desired)
[web1]     @@ -1,7 +1 @@
[web1]     -
[web1]     -The programs included with the Debian GNU/Linux system are free software;
[web1]     -the exact distribution terms for each program are described in the
[web1]     -individual files in /usr/share/doc/*/copyright.
[web1]     -
[web1]     -Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
[web1]     -permitted by applicable law.
[web1]     +Welcome to web1. This server is managed by sadame.
[web1]
[web1] Plan: 0 to create, 1 to update, 0 to remove, 0 to run, 0 unchanged.

Hosts
  web1  ok

plan on 1 host: 1 ok. Finished in 0.22s.
```

Three things are new in this output:

- The first line, `copying sadame-linux-arm64 to ~/.sadame/bin/…`: sadame copied **itself** to the server. It does that once per version; the directory is named after the binary's SHA-256 checksum, which is checked after the copy.
- Every line from the server starts with `[web1]`, its name.
- After the server's own output, a table says how each host did, and the last line sums it up.

Sadame connected with your own `ssh` command, so everything in `~/.ssh/config` applies. On the practice machine, that file says to log in to the lab's servers as `deploy`; on your own computer, it is where you keep ports, keys and jump hosts. You did not tell sadame any of that, and did not need to.

10.2. Permission to change things

The plan worked because reading `/etc/motd` needs no special rights. Writing it does. Apply without asking for root:

```
$ sadame apply motd.rb -h web1
[web1] sadame 0.5.1 apply on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] failed     file[/etc/motd]
[web1]            --- /etc/motd
[web1]            +++ /etc/motd (desired)
[web1]            @@ -1,7 +1 @@
[web1]            -
[web1]            -The programs included with the Debian GNU/Linux system are free software;
[web1]            -the exact distribution terms for each program are described in the
[web1]            -individual files in /usr/share/doc/*/copyright.
[web1]            -
[web1]            -Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
[web1]            -permitted by applicable law.
[web1]            +Welcome to web1. This server is managed by sadame.
[web1]            error: RuntimeError: open /etc/motd.sadame-new
[web1]
[web1] Summary
[web1]   changed      0
[web1]   unchanged    0
[web1]   failed       1  file[/etc/motd]
[web1]   skipped      0
[web1]   notified     0
[web1]
[web1] Finished in 0.00s.

Hosts
  web1  failed

apply on 1 host: 1 failed. Finished in 0.21s.

$ echo $?
1
```

The user `deploy` may not create a file in `/etc`, so sadame could not open the temporary file it writes the new content to (`open /etc/motd.sadame-new`), and the resource failed. Nothing was changed, and the exit status is `1`.

`--sudo` runs sadame on the server through `sudo`, which `deploy` may use without a password:

```
$ sadame apply motd.rb -h web1 --sudo
[web1] sadame 0.5.1 apply on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] changed    file[/etc/motd]
[web1]            --- /etc/motd
[web1]            +++ /etc/motd (desired)
[web1]            @@ -1,7 +1 @@
[web1]            -
[web1]            -The programs included with the Debian GNU/Linux system are free software;
[web1]            -the exact distribution terms for each program are described in the
[web1]            -individual files in /usr/share/doc/*/copyright.
[web1]            -
[web1]            -Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
[web1]            -permitted by applicable law.
[web1]            +Welcome to web1. This server is managed by sadame.
```

```
[web1]
[web1] Summary
[web1]   changed      1  file[/etc/motd]
[web1]   unchanged    0
[web1]   failed       0
[web1]   skipped      0
[web1]   notified     0
[web1]
[web1] Finished in 0.00s.

Hosts
  web1  ok

apply on 1 host: 1 ok. Finished in 0.22s.

$ ssh web1 cat /etc/motd
Welcome to web1. This server is managed by sadame.
```

The binary was already there, so no copying this time.



`--sudo` needs **passwordless** sudo for the user you log in as, because nobody is there to type a password. That is the usual setup for deployment users on cloud servers. Chapter 20 shows the message you get when sudo asks for a password.

10.3. Several servers at once

`-h` takes a comma-separated list:

```
$ sadame apply motd.rb -h web1,web2 --sudo
[web2] copying sadame-linux-arm64 to
~/.sadame/bin/dcf637ae7d2f79587cd36cc4fb47eff845444424933647a5e75565917475c21d/
[web1] sadame 0.5.1 apply on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] unchanged  file[/etc/motd]
[web1]
[web1] Summary
[web1]   changed      0
[web1]   unchanged    1
[web1]   failed       0
[web1]   skipped      0
[web1]   notified     0
[web1]
[web1] Finished in 0.00s.
[web2] sadame 0.5.1 apply on web2 (Debian GNU/Linux 12 (bookworm))
[web2]
[web2] changed    file[/etc/motd]
[web2]            --- /etc/motd
[web2]            +++ /etc/motd (desired)
[web2]            @@ -1,7 +1 @@
[web2]            -
[web2]            -The programs included with the Debian GNU/Linux system are free software;
[web2]            -the exact distribution terms for each program are described in the
[web2]            -individual files in /usr/share/doc/*/copyright.
[web2]            -
[web2]            -Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
[web2]            -permitted by applicable law.
[web2]            +Welcome to web2. This server is managed by sadame.
[web2]
[web2] Summary
[web2]   changed      1  file[/etc/motd]
[web2]   unchanged    0
[web2]   failed       0
[web2]   skipped      0
[web2]   notified     0
[web2]
[web2] Finished in 0.00s.
```

```
Hosts
  web1  ok
  web2  ok

apply on 2 hosts: 2 ok. Finished in 0.24s.
```

Both servers ran at the same time, each in its own process. web2 got the binary copied first; web1 had it and its message already, so it reports `unchanged`. The table at the end is what you read first on a fleet: one line per host, with `ok`, `failed`, `drift`, or `unreachable`.

10.4. What happens on the server

For every run, on every host, sadame:

1. connects with `ssh` and asks the server's system and architecture with `uname`;
2. with `--sudo`, checks that sudo works without a password, before anything is copied;
3. copies the right sadame binary there if it is not there yet, and checks its checksum;
4. packs the recipe's directory, templates and files included, copies it over, and unpacks it in a fresh directory under `~/.sadame/run/`;
5. runs sadame there, with `sudo` if asked, and streams its output back;
6. removes the recipe's directory.

What stays on the server is the binary in `~/.sadame/bin/`, reused by the next run. No agent, no daemon, no open port. To remove every trace, delete `~/.sadame`.

The servers need SSH, a shell, `tar`, and `sha256sum` or `shasum`. If a server has a different architecture from your computer, say your laptop is an ARM Mac and the server x86_64, sadame downloads the matching Linux binary from its GitHub release once, checks it, and keeps it under `~/.sadame/dist/` on your computer.

What you learned

- `-h HOST` runs a recipe on a server over SSH; `-h a,b` on several, in parallel.
- `~/.ssh/config` applies: users, ports, keys and jump hosts.
- `--sudo` runs sadame with passwordless sudo, for recipes that change the system.
- Sadame copies itself to `~/.sadame/bin/` once per version and leaves nothing else behind.

Try it

1. Run `sadame plan motd.rb -h db1`. The Rocky Linux server has no `/etc/motd` content yet; what does the diff look like?
2. Run `ssh web1 ls -la .sadame/bin` to see the copied binary.
3. Apply to all three servers with one command.

Chapter 11. The Inventory

Typing host names on every command does not scale past a handful of servers, and it says nothing about **what** each server is. An inventory does: a file that lists your servers, how to reach them, which groups they belong to, and the values that differ between them. Sadame's inventory is `hosts.rb`, and like everything else it is Ruby.

```
$ cd ~/code/11-inventory
```

11.1. hosts.rb

code/11-inventory/hosts.rb

```
# The lab's servers, in groups, with vars for a group and for one host.
group :web do
  host "web1", user: "deploy", sudo: true
  host "web2", user: "deploy", sudo: true, vars: { tier: "canary" }
  vars role: "web server"
end

group :db do
  host "db1", user: "deploy", sudo: true
  vars role: "database server"
end

group :all, include: [:web, :db] do
  vars environment: "lab"
end
```

Three methods describe everything:

`group :name do … end`

: A named set of hosts. A group can also take in other groups with `include:`, as `all` does here: `all` is whatever `web` and `db` hold, without listing any server twice.

`host "name", options`

: A server, with how to reach it: `user:`, `port:`, `address:` when the name is only a label, `sudo:`, and `ssh_opts:` for extra SSH options. And `vars:` for values that are this host's alone.

`vars key: value`

: Values for every host of the group.

`sudo: true` saves you from typing `--sudo`: every run on these hosts uses sudo.

11.2. Seeing what the inventory says

`sadame hosts` prints the selected hosts with everything they would get, without connecting to any:

```
$ sadame hosts --limit web2
sadame 0.5.1 hosts: 1 host from hosts.rb

web2
  address   web2
  user      deploy
```

```
  port      (ssh default)
  sudo      yes
  ssh_opts  (none)
  groups    web, all
  vars
    role         "web server"       group web
    environment  "lab"              group all
    tier         "canary"           host
    contact      "ops@example.com"  vars.rb
```

The `vars` section is the most useful part: every var the host's recipes will see, and **where it came from**. web2's `tier` comes from the host, `role` from the group `web`, `environment` from the group `all`, and `contact` from `vars.rb`. When a recipe does something you did not expect, this is where to look first.

11.3. Choosing hosts

With an inventory, you choose hosts by group with `-g`, and narrow any selection with `--limit` and a pattern:

Command	Runs on
`sadame plan recipe.rb -g web`	web1 and web2
`sadame plan recipe.rb -g web,db`	all three
`sadame plan recipe.rb -g all --limit "web*"`	web1 and web2
`sadame plan recipe.rb --limit web2`	web2
`sadame plan recipe.rb -h db1`	db1, with its settings from `hosts.rb`

The recipe builds the login message from the vars, and from the host's name and groups, which it hands to the template with `locals:`:

code/11-inventory/recipe.rb

```
# A login message built from the host's name, its groups and its vars.
file "/etc/motd",
     template: "motd.erb",
     locals: { name: host_name, groups: host_groups },
     mode: "0644"
```

code/11-inventory/templates/motd.erb

```
<%= locals[:name] %> is a <%= vars[:role] %> in the <%= vars[:environment] %> environment (<%= vars[:tier] %>
tier).
Groups: <%= locals[:groups].join(", ") %>
Questions: <%= vars[:contact] %>
```

`host_groups` lists the groups a host belongs to, including those that include them: web1 is in `web` and in `all`. Plan the web servers:

```
$ sadame plan recipe.rb -g web
[web1] sadame 0.5.1 plan on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web2] sadame 0.5.1 plan on web2 (Debian GNU/Linux 12 (bookworm))
[web2]
[web1] ~ file[/etc/motd]
```

```
[web1]     --- /etc/motd
[web1]     +++ /etc/motd (desired)
[web1]     @@ -1 +1,3 @@
[web1]     -Welcome to web1. This server is managed by sadame.
[web1]     +web1 is a web server in the lab environment (stable tier).
[web1]     +Groups: web, all
[web1]     +Questions: ops@example.com
[web1]
[web1] Plan: 0 to create, 1 to update, 0 to remove, 0 to run, 0 unchanged.
[web2] ~ file[/etc/motd]
[web2]     --- /etc/motd
[web2]     +++ /etc/motd (desired)
[web2]     @@ -1 +1,3 @@
[web2]     -Welcome to web2. This server is managed by sadame.
[web2]     +web2 is a web server in the lab environment (canary tier).
[web2]     +Groups: web, all
[web2]     +Questions: ops@example.com
[web2]
[web2] Plan: 0 to create, 1 to update, 0 to remove, 0 to run, 0 unchanged.

Hosts
  web1  ok
  web2  ok

plan on 2 hosts: 2 ok. Finished in 0.22s.
```

Same recipe, two different messages: web2 is the `canary` tier because its host vars say so.

11.4. All of them

Apply to the `all` group:

```
$ sadame apply recipe.rb -g all
[web1] sadame 0.5.1 apply on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web2] sadame 0.5.1 apply on web2 (Debian GNU/Linux 12 (bookworm))
[web2]
[db1] copying sadame-linux-arm64 to
~/.sadame/bin/dcf637ae7d2f79587cd36cc4fb47eff845444424933647a5e75565917475c21d/
[web1] changed    file[/etc/motd]
[web1]            --- /etc/motd
[web1]            +++ /etc/motd (desired)
[web1]            @@ -1 +1,3 @@
[web1]            -Welcome to web1. This server is managed by sadame.
[web1]            +web1 is a web server in the lab environment (stable tier).
[web1]            +Groups: web, all
[web1]            +Questions: ops@example.com
[web1]
[web1] Summary
[web1]   changed      1  file[/etc/motd]
[web1]   unchanged    0
[web1]   failed       0
[web1]   skipped      0
[web1]   notified     0
[web1]
[web1] Finished in 0.00s.
[web2] changed    file[/etc/motd]
[web2]            --- /etc/motd
[web2]            +++ /etc/motd (desired)
[web2]            @@ -1 +1,3 @@
[web2]            -Welcome to web2. This server is managed by sadame.
[web2]            +web2 is a web server in the lab environment (canary tier).
[web2]            +Groups: web, all
[web2]            +Questions: ops@example.com
[web2]
[web2] Summary
[web2]   changed      1  file[/etc/motd]
[web2]   unchanged    0
[web2]   failed       0
[web2]   skipped      0
[web2]   notified     0
```

```
[web2]
[web2] Finished in 0.00s.
[db1] sadame 0.5.1 apply on db1 (Rocky Linux 9.8 (Blue Onyx))
[db1]
[db1] changed    file[/etc/motd]
[db1]            --- /etc/motd
[db1]            +++ /etc/motd (desired)
[db1]            @@ -0,0 +1,3 @@
[db1]            +db1 is a database server in the lab environment (stable tier).
[db1]            +Groups: db, all
[db1]            +Questions: ops@example.com
[db1]
[db1] Summary
[db1]   changed      1  file[/etc/motd]
[db1]   unchanged    0
[db1]   failed       0
[db1]   skipped      0
[db1]   notified     0
[db1]
[db1] Finished in 0.01s.

Hosts
  web1  ok
  web2  ok
  db1   ok

apply on 3 hosts: 3 ok. Finished in 0.28s.

$ ssh web2 cat /etc/motd
web2 is a web server in the lab environment (canary tier).
Groups: web, all
Questions: ops@example.com
```

Look at how the lines arrive: web1's and web2's output is interleaved, because both ran at the same time and printed as they went. The `[name]` prefix keeps every line attributable, and the table at the end puts the results in order. db1 got its binary copied on its first contact.

11.5. Overriding a var once

`--var` still wins over everything, which makes "what if" questions cheap:

```
$ sadame plan recipe.rb -g db --var environment=staging
[db1] sadame 0.5.1 plan on db1 (Rocky Linux 9.8 (Blue Onyx))
[db1]
[db1] ~ file[/etc/motd]
[db1]     --- /etc/motd
[db1]     +++ /etc/motd (desired)
[db1]     @@ -1,3 +1,3 @@
[db1]     -db1 is a database server in the lab environment (stable tier).
[db1]     +db1 is a database server in the staging environment (stable tier).
[db1]      Groups: db, all
[db1]      Questions: ops@example.com
[db1]
[db1] Plan: 0 to create, 1 to update, 0 to remove, 0 to run, 0 unchanged.

Hosts
  db1  ok

plan on 1 host: 1 ok. Finished in 0.22s.
```

The full order, from weakest to strongest: `vars.rb`, the vars of the host's groups (a group that includes another wins over it), the host's own vars, and `--var`.

11.6. Inventories from somewhere else

Because `hosts.rb` is Ruby, it can build the list from anything: a JSON file kept by another team, or your cloud provider's command-line tool. `command_output` runs a command on your computer and returns its output:

```
servers = JSON.parse(command_output("hcloud server list -o json"))
group :web do
  servers.select { |s| s["labels"]["role"] == "web" }.each do |s|
    host s["name"], address: s["public_net"]["ipv4"]["ip"], user: "deploy", sudo: true
  end
end
```

Sadame's documentation has complete examples for Hetzner Cloud, DigitalOcean and AWS.



`hosts.rb` runs on **your** computer, not on the servers, and its vars are sent to every host it selects. Keep secrets out of it; read them from the environment (`ENV["DB_PASSWORD"]`) as chapter 15 does.

What you learned

- `hosts.rb` lists hosts in groups, with connection options and vars; groups can include other groups.
- `sadame hosts` shows each host's settings and where every var came from.
- `-g` selects groups, `--limit` narrows by name, `-h` picks hosts.
- Vars layer: `vars.rb`, group vars, host vars, `--var`.
- `host_groups` and `host_name` tell a recipe which machine it runs on.

Try it

1. Give `db1` its own `vars: { tier: "primary" }` and run `sadame hosts --limit db1`. Where does `tier` come from now?
2. Add a group `:canary` with only web2 in it, and plan with `-g canary`.
3. Run `sadame hosts -g all --json` and find web1's vars in the output.

Chapter 12. Facts and Platforms

Servers differ. One runs Debian and calls its cron package `cron`, the other runs Rocky Linux and calls it `cronie`. One has two CPUs and the other sixteen, and an application's worker count should follow. A recipe that hardcodes either difference works on one machine and is wrong on the next.

Sadame gives a recipe two ways to see the machine it runs on: `platform`, which says which Linux it is, and `facts`, which says what the machine has.

```
$ cd ~/code/12-facts-and-platforms
```

12.1. What a machine looks like

`sadame facts` prints what a recipe would see. Ask the database server:

```
$ sadame facts -h db1
[db1] sadame 0.5.1 facts on db1 (Rocky Linux 9.8 (Blue Onyx))
[db1]
[db1] hostname           db1
[db1] fqdn               db1
[db1] arch               aarch64
[db1] kernel             7.2.12-orbstack-00394-gbecdd6130068
[db1] cpus               10
[db1] memory_mb          8003
[db1] swap_mb            9027
[db1] root_disk_gb       139
[db1] ipv4               192.168.107.4
[db1] default_interface  eth0
[db1] ipv4_addresses     192.168.107.4
[db1] ipv6_addresses     (none)
[db1] virtualization     container-other
[db1] container          true
[db1] service_manager    systemd
[db1] machine_id         59b401651f4b47e984b652e719de73e7

Hosts
  db1  ok

facts on 1 host: 1 ok. Finished in 0.22s.
```

Sixteen facts, all read on db1 itself: its name, CPUs, memory, disk, addresses, and what kind of machine it is. A few are worth a second look:

`cpus` **and** `memory_mb`

: The CPUs sadame may use and the memory, in megabytes. In the lab they are the numbers of the computer running Docker: containers share its kernel and see its resources. On a virtual machine or a cloud server you would see that machine's own.

`ipv4`

: The address the machine would use to reach the internet. Sadame works it out from the routing table and sends nothing.

`virtualization` **and** `container`

: What the machine runs on. `container` is `true` here, which a recipe can use to leave out what makes no sense in a container, such as kernel tuning.

`service_manager`

: Which init system the `service` resource will talk to: `systemd` here, `openrc` on Alpine.

A recipe reads a fact with `facts[:name]`. Each fact is read the first time a recipe asks for it, so facts cost nothing until you use them, and a misspelled one, `facts[:cpu]`, is an error that lists the right names instead of a silent `nil`.

12.2. The platform

`platform` describes the operating system, from `/etc/os-release`:

Method	Example values
`platform.family`	`"debian"` (Debian, Ubuntu), `"rhel"` (Rocky, Alma, Fedora), `"alpine"`, `"arch"`
`platform.package_manager`	`"apt"`, `"dnf"`, `"apk"`, `"pacman"`
`platform.id`	`"debian"`, `"rocky"`, `"ubuntu"`
`platform.version`	`"12"`, `"9.4"`
`platform.name`	`"Debian GNU/Linux 12 (bookworm)"`

Use `family` for most decisions. Debian and Ubuntu share package names and file locations far more often than Debian and Rocky do, and `family` groups them the way their packages are built.

12.3. One recipe, two families

code/12-facts-and-platforms/recipe.rb

```
# One recipe for Debian and Rocky Linux: the names that differ come from
# the platform, the sizes from the machine's facts.

# cron is called cron on Debian and cronie on Rocky, and so is its service:
# cron and crond.
cron_package, cron_service = if platform.family == "rhel"
                               ["cronie", "crond"]
                             else
                               ["cron", "cron"]
                             end

package cron_package
service cron_service, enable: true, state: :running

# Two workers per CPU, an eighth of the memory for a cache.
file "/etc/app-limits.conf",
     content: "workers = #{facts[:cpus] * 2}\ncache_mb = #{facts[:memory_mb] / 8}\n",
     mode: "0644"
```

The recipe picks the package and service names with an ordinary Ruby `if` that returns two values, then declares them like any other. The `file` resource computes its numbers from facts: two workers per CPU, an eighth of the memory for a cache.

This chapter has its own `hosts.rb` with the same three servers as chapter 11, without the vars. Plan on all of them:

```
$ sadame plan recipe.rb -g all
```

```
[web2] sadame 0.5.1 plan on web2 (Debian GNU/Linux 12 (bookworm))
[web2]
[web1] sadame 0.5.1 plan on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] + package[cron]
[web1]     state: absent -> installed
[web1]
[web2] + package[cron]
[web2]     state: absent -> installed
[web2]
[web1] ~ service[cron]
[web1]     enabled: false -> true
[web1]     state: stopped -> running
[web1]
[web2] ~ service[cron]
[web2]     enabled: false -> true
[web2]     state: stopped -> running
[web2]
[web2] + file[/etc/app-limits.conf]
[web1] + file[/etc/app-limits.conf]
[web1]     state: absent -> present
[web1]     mode: 0644
[web1]     --- /dev/null
[web1]     +++ /etc/app-limits.conf (desired)
[web1]     @@ -0,0 +1,2 @@
[web1]     +workers = 20
[web1]     +cache_mb = 1000
[web1]
[web1] Plan: 2 to create, 1 to update, 0 to remove, 0 to run, 0 unchanged.
[web2]     state: absent -> present
[web2]     mode: 0644
[web2]     --- /dev/null
[web2]     +++ /etc/app-limits.conf (desired)
[web2]     @@ -0,0 +1,2 @@
[web2]     +workers = 20
[web2]     +cache_mb = 1000
[web2]
[web2] Plan: 2 to create, 1 to update, 0 to remove, 0 to run, 0 unchanged.
[db1] sadame 0.5.1 plan on db1 (Rocky Linux 9.8 (Blue Onyx))
[db1]
[db1] + package[cronie]
[db1]     state: absent -> installed
[db1]
[db1] ~ service[crond]
[db1]     enabled: false -> true
[db1]     state: stopped -> running
[db1]
[db1] + file[/etc/app-limits.conf]
[db1]     state: absent -> present
[db1]     mode: 0644
[db1]     --- /dev/null
[db1]     +++ /etc/app-limits.conf (desired)
[db1]     @@ -0,0 +1,2 @@
[db1]     +workers = 20
[db1]     +cache_mb = 1000
[db1]
[db1] Plan: 2 to create, 1 to update, 0 to remove, 0 to run, 0 unchanged.

Hosts
  web1  ok
  web2  ok
  db1   ok

plan on 3 hosts: 3 ok. Finished in 0.27s.
```

The Debian servers get `cron`, db1 gets `cronie` and `crond`. All three get the same limits, because in the lab they all see the same computer. Apply, and check one:

```
$ sadame apply recipe.rb -g all > apply.log; echo $?
0

$ ssh db1 cat /etc/app-limits.conf
```

```
workers = 20
cache_mb = 1000
```

The apply's output went to `apply.log`; `echo $?` printed its exit status, `0`, which says everything went well. On a fleet, that habit of keeping the full log and looking at the status first saves a lot of scrolling.

12.4. Facts follow the machine

Facts are read on every run. If db1 gets more memory, the next `plan` shows `/etc/app-limits.conf` with a new `cache_mb`, and `drift` reports the file as different until you apply. A configuration computed from facts stays sized to its machine without anybody editing a number.



Facts are for what a machine **has**. What you **decide** about a machine, such as its role or its tier, belongs in `hosts.rb` and vars. `hosts.rb` runs on your computer and cannot see facts at all.

12.5. When the platform is not supported

`platform.family` is `nil` on a system sadame does not know. `package` fails there with `no supported package manager`, and `service` fails on a machine where neither systemd nor OpenRC runs, but `file`, `directory`, `link` and the others work everywhere. If a recipe must run on such a machine, test for `nil` and decide in the recipe what to do.

What you learned

- `sadame facts` shows what recipes see on a machine; `facts[:name]` reads one in a recipe or a template.
- `platform.family` tells Debian-like from Red Hat-like systems; use it for package and service names.
- Facts are read every run, so configuration computed from them follows the machine.
- Decide in `hosts.rb`; measure with `facts`.

Try it

1. Run `sadame facts -h web1 --json` and find `memory_mb` in the output.
2. Change the recipe so a container gets only one worker: `facts[:container] ? 1 : facts[:cpus] * 2`. Plan it.
3. Add `package platform.family == "rhel" ? "vim-enhanced" : "vim"` and apply to all three servers.

Chapter 13. Organizing a Growing Recipe

One file is right for a small recipe. As soon as servers do different jobs, one file becomes a long list of `if host_groups.include?(…)` blocks that nobody wants to read. This chapter splits a recipe into **roles**, one file for each job a server can have, and shows how to run only part of it with **tags**.

```
$ cd ~/code/13-organizing
```

13.1. The layout

```
$ find . -type f | sort
./hosts.rb
./roles/base.rb
./roles/db.rb
./roles/templates/motd.erb
./roles/web.rb
./site.rb
./vars.rb
```

This is the layout the rest of the book uses, and a good one for your own servers:

`site.rb`

: The entry point. It says which roles each server gets, and nothing else.

`roles/`

: One file per role: what **every** server gets (`base`), what a web server gets, what a database server gets. A role's templates live next to it, in `roles/templates/`.

`hosts.rb` **and** `vars.rb`

: The inventory and the defaults, as before.

Nothing about this layout is required. Sadame only cares about the file you give it and the files that file includes.

13.2. include_recipe

code/13-organizing/site.rb

```
# The entry point: a base for every server, then a role per group. Each
# include_recipe tags what the role declares, so --tags web runs only the
# web role.
include_recipe "roles/base", tags: [:base]
include_recipe "roles/web", tags: [:web] if host_groups.include?(:web)
include_recipe "roles/db", tags: [:db] if host_groups.include?(:db)
```

`include_recipe "roles/web"` reads `roles/web.rb` and adds its resources at that point, in order, as if they were written there. The path is relative to the file that contains the call, and `.rb` is optional. Because `site.rb` is Ruby, the decision of which roles a server gets is a plain `if` on its groups.

The roles are short:

code/13-organizing/roles/base.rb

```
# What every server gets.
file "/etc/motd", template: "motd.erb", mode: "0644"

# The time zone: /etc/localtime is a link into the zoneinfo database.
link "/etc/localtime", to: "/usr/share/zoneinfo/#{vars[:timezone]}", force: true
```

code/13-organizing/roles/web.rb

```
# The web servers: nginx, running.
package "nginx"
service "nginx", enable: true, state: :running
```

code/13-organizing/roles/db.rb

```
# The database server: SQLite's command-line tool, under its name for each
# family.
package platform.family == "rhel" ? "sqlite" : "sqlite3"
```

`roles/base.rb` names its template `motd.erb`, and sadame finds it in `roles/templates/`: templates and source files are looked up next to the file that uses them, not next to `site.rb`. A role directory carries its own templates, which makes it easy to move or share.

A file included twice is read only the first time. Two roles can both include a shared file, say `roles/common/users.rb`, without declaring its resources twice.

13.3. Tags

Each `include_recipe` in `site.rb` has `tags:`. Every resource the included file declares gets those tags, and `--tags` selects resources by tag. Plan only the base role:

```
$ sadame plan site.rb -g all --tags base
[web2] sadame 0.5.1 plan on web2 (Debian GNU/Linux 12 (bookworm))
[web2] Only resources tagged base: 2 of 4.
[web2]
[web1] sadame 0.5.1 plan on web1 (Debian GNU/Linux 12 (bookworm))
[web1] Only resources tagged base: 2 of 4.
[web1]
[web2] ~ file[/etc/motd]
[web2]     --- /etc/motd
[web2]     +++ /etc/motd (desired)
[web2]     @@ -1,3 +1 @@
[web2]     -web2 is a web server in the lab environment (canary tier).
[web2]     -Groups: web, all
[web2]     -Questions: ops@example.com
[web2]     +Managed by sadame. Questions: ops@example.com
[web2]
[web1] ~ file[/etc/motd]
[web1]     --- /etc/motd
[web1]     +++ /etc/motd (desired)
[web1]     @@ -1,3 +1 @@
[web1]     -web1 is a web server in the lab environment (stable tier).
[web1]     -Groups: web, all
[web1]     -Questions: ops@example.com
[web1]     +Managed by sadame. Questions: ops@example.com
[web1]
[web2] ~ link[/etc/localtime]
[web2]     to: /usr/share/zoneinfo/Etc/UTC -> /usr/share/zoneinfo/Europe/Istanbul
[web2]
[web2] Plan: 0 to create, 2 to update, 0 to remove, 0 to run, 0 unchanged.
```

```
[web1] ~ link[/etc/localtime]
[web1]     to: /usr/share/zoneinfo/Etc/UTC -> /usr/share/zoneinfo/Europe/Istanbul
[web1]
[web1] Plan: 0 to create, 2 to update, 0 to remove, 0 to run, 0 unchanged.
[db1] sadame 0.5.1 plan on db1 (Rocky Linux 9.8 (Blue Onyx))
[db1] Only resources tagged base: 2 of 3.
[db1]
[db1] ~ file[/etc/motd]
[db1]     --- /etc/motd
[db1]     +++ /etc/motd (desired)
[db1]     @@ -1,3 +1 @@
[db1]     -db1 is a database server in the lab environment (stable tier).
[db1]     -Groups: db, all
[db1]     -Questions: ops@example.com
[db1]     +Managed by sadame. Questions: ops@example.com
[db1]
[db1] ~ link[/etc/localtime]
[db1]     to: /usr/share/zoneinfo/Etc/UTC -> /usr/share/zoneinfo/Europe/Istanbul
[db1]
[db1] Plan: 0 to create, 2 to update, 0 to remove, 0 to run, 0 unchanged.

Hosts
  web1  ok
  web2  ok
  db1   ok

plan on 3 hosts: 3 ok. Finished in 0.25s.
```

The header line says what was selected: two of four resources on the web servers, two of three on db1. The packages and services were not read at all. That makes tags useful in three situations:

- **Speed**: change the login message on a hundred servers without waiting for every package manager.
- **Focus**: review a plan for one role without the noise of the others.
- **Caution**: `--skip-tags db` when the database servers must not be touched today.

You can also tag single resources: `package "nginx", tags: [:nginx]`. A resource's tags are its own plus those of every `include_recipe` it came through.



With `--tags`, resources without any tag are never selected. And a notification still runs when its target is not selected: changing a tagged configuration file still reloads the service it notifies. Keep scheduled drift checks **without** tags, so they cover everything.

Now apply everything:

```
$ sadame apply site.rb -g all > apply.log; echo $?
0

$ grep -E '^\[[a-z0-9]+\] changed ' apply.log
[web1] changed    file[/etc/motd]
[web2] changed    file[/etc/motd]
[web1] changed    link[/etc/localtime]
[web2] changed    link[/etc/localtime]
[db1] changed    file[/etc/motd]
[db1] changed    link[/etc/localtime]
[db1] changed    package[sqlite]
[web1] changed    package[nginx]
[web1] changed    service[nginx]
[web2] changed    package[nginx]
[web2] changed    service[nginx]
```

```
$ tail -n 6 apply.log
Hosts
  web1  ok
  web2  ok
  db1   ok

apply on 3 hosts: 3 ok. Finished in 4.39s.
```

The full output went to `apply.log`; `grep` picks the changes out of it. The web servers got nginx, db1 got SQLite under its Rocky name, and everyone got the new login message and the Istanbul time zone. The lines of the three hosts are interleaved because they ran at the same time; the `[name]` prefix keeps them apart.

13.4. Starting a project with sadame init

When you start your own project, `sadame init` writes a working starting point:

```
$ sadame init myinfra
created myinfra/recipe.rb
created myinfra/vars.rb
created myinfra/hosts.rb
created myinfra/templates/motd.erb

Next:
  sadame validate myinfra        check the recipe without connecting to anything
  sudo sadame plan myinfra       see what it would change on this machine
  sadame plan myinfra -g all     on your servers, once myinfra/hosts.rb lists them

$ find myinfra -type f | sort
myinfra/hosts.rb
myinfra/recipe.rb
myinfra/templates/motd.erb
myinfra/vars.rb

$ sadame validate myinfra
sadame 0.5.1 validate myinfra/recipe.rb, as on Debian GNU/Linux 12 (bookworm)

  local  ok     1 resource

validate on 1 host: 1 ok.
```

Four files, the same shape as this chapter: a recipe, defaults, an inventory with an example group, and a template. It writes nothing if any of those files already exists, so it cannot overwrite your work. The `Next:` lines are the three commands to run after it.

`sadame validate` checks a recipe without connecting to anything: it loads it as on a Debian machine and reports mistakes. Chapter 19 is about it.

What you learned

- Split recipes into roles: `site.rb` decides, `roles/*.rb` declare.
- `include_recipe` adds a file's resources in place; templates are found next to the file that uses them.
- `tags:` on `include_recipe` or a resource, and `--tags` and `--skip-tags` on the command line, run part of a recipe.
- `sadame init` starts a project; `sadame validate` checks one offline.

Try it

1. Run `sadame plan site.rb -g all --skip-tags base`. What is left?
2. Add a role `roles/tools.rb` that installs `htop`, include it for every server with `tags: [:tools]`, and apply only that role.
3. Run `sadame init` in an empty directory and read the four files it writes.

Part IV: Projects

Chapter 14. Project: A Website

Part IV puts everything together in projects that build on each other: a website, a database server, an application, and finally the whole fleet. Each starts from a goal, writes the recipe, and runs it on the lab's servers.

The goal of this one: both web servers serve the same small site with nginx, from `/var/www/lab-site`, and anyone who edits the site's files by hand will see it reported.

```
$ cd ~/code/14-website
```

14.1. How Debian's nginx finds sites

Debian's nginx package reads every file in `/etc/nginx/sites-enabled/`. The convention is to keep site configurations in `sites-available/` and switch a site on with a symbolic link from `sites-enabled/`, which makes switching a site off a matter of removing a link. The package ships a site called `default`, the "Welcome to nginx!" page, which is switched on.

So the recipe needs: nginx; the site's directory and page; its configuration in `sites-available/`; a link to switch it on; the `default` link gone; nginx running, and reloaded whenever any of that configuration changes.

14.2. The recipe

code/14-website/roles/web.rb

```
# nginx serving one site from /var/www/<site_name>, reloaded when its
# configuration changes. Debian's nginx reads the sites linked from
# sites-enabled/; the package's default site is switched off.
site = vars[:site_name]
root = "/var/www/#{site}"

package "nginx"

directory root, mode: "0755"
file "#{root}/index.html",
     template: "index.html.erb",
     locals: { host: host_name },
     mode: "0644"

file "/etc/nginx/sites-available/#{site}",
     template: "site.conf.erb",
     locals: { root: root },
     mode: "0644",
     notifies: [:reload, "service[nginx]"]

link "/etc/nginx/sites-enabled/#{site}",
     to: "/etc/nginx/sites-available/#{site}",
     notifies: [:reload, "service[nginx]"]

link "/etc/nginx/sites-enabled/default",
     state: :absent,
     notifies: [:reload, "service[nginx]"]

service "nginx", enable: true, state: :running
```

code/14-website/vars.rb

```
{
  site_name: "lab-site",
  site_title: "Welcome to the lab",
  server_name: "_"
}
```

The two templates use vars for what is the same on every server and `locals:` for what the recipe computes:

code/14-website/roles/templates/site.conf.erb

```
# Managed by sadame.
server {
    listen 80 default_server;
    server_name <%= vars[:server_name] %>;
    root <%= locals[:root] %>;
    index index.html;
}
```

code/14-website/roles/templates/index.html.erb

```
<!doctype html>
<title><%= vars[:site_title] %></title>
<h1><%= vars[:site_title] %></h1>
<p>Served by <%= locals[:host] %>.</p>
```

`server_name _;` is nginx's way of saying "any name", which suits a server reached by address. `site.rb` includes the role with the tag `web`, and `hosts.rb` lists web1 and web2 in the group `web`.

14.3. Plan, apply, look

Plan on one server first:

```
$ sadame plan site.rb -g web --limit web1
[web1] sadame 0.5.1 plan on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] + directory[/var/www/lab-site]
[web1]     state: absent -> present
[web1]     mode: 0755
[web1]
[web1] + file[/var/www/lab-site/index.html]
[web1]     state: absent -> present
[web1]     mode: 0644
[web1]     --- /dev/null
[web1]     +++ /var/www/lab-site/index.html (desired)
[web1]     @@ -0,0 +1,4 @@
[web1]     +<!doctype html>
[web1]     +<title>Welcome to the lab</title>
[web1]     +<h1>Welcome to the lab</h1>
[web1]     +<p>Served by web1.</p>
[web1]
[web1] + file[/etc/nginx/sites-available/lab-site]
[web1]     state: absent -> present
[web1]     mode: 0644
[web1]     --- /dev/null
[web1]     +++ /etc/nginx/sites-available/lab-site (desired)
[web1]     @@ -0,0 +1,7 @@
[web1]     +# Managed by sadame.
[web1]     +server {
[web1]     +    listen 80 default_server;
[web1]     +    server_name _;
[web1]     +    root /var/www/lab-site;
[web1]     +    index index.html;
```

```
[web1]     +}
[web1]     notifies: reload service[nginx] (delayed)
[web1]
[web1] + link[/etc/nginx/sites-enabled/lab-site]
[web1]     state: absent -> present
[web1]     to: /etc/nginx/sites-available/lab-site
[web1]     notifies: reload service[nginx] (delayed)
[web1]
[web1] - link[/etc/nginx/sites-enabled/default]
[web1]     state: present -> absent
[web1]     notifies: reload service[nginx] (delayed)
[web1]
[web1] Plan: 4 to create, 0 to update, 1 to remove, 0 to run, 2 unchanged.

Hosts
  web1  ok

plan on 1 host: 1 ok. Finished in 0.23s.
```

`2 unchanged`: nginx is installed and running since chapter 13. Three resources would notify the reload. Apply on both:

```
$ sadame apply site.rb -g web > apply.log; echo $?
0

$ grep -E '^\[web1\] (changed|notified) ' apply.log
[web1] changed    directory[/var/www/lab-site]
[web1] changed    file[/var/www/lab-site/index.html]
[web1] changed    file[/etc/nginx/sites-available/lab-site]
[web1] changed    link[/etc/nginx/sites-enabled/lab-site]
[web1] changed    link[/etc/nginx/sites-enabled/default]
[web1] notified   service[nginx]: reload (delayed, from file[/etc/nginx/sites-available/lab-site])

$ tail -n 6 apply.log

Hosts
  web1  ok
  web2  ok

apply on 2 hosts: 2 ok. Finished in 0.24s.
```

Three resources changed and asked for a reload, and nginx reloaded **once**, at the end. The `notified` line names the first resource that asked. Delayed notifications to the same resource with the same action collapse into one: the service is reloaded with the final configuration, not three times on the way there.

`systemctl reload nginx` returns as soon as nginx has been told to reload; nginx then starts new workers with the new configuration in the background. Give it a moment, then look:

```
$ curl -s http://web1/
<!doctype html>
<title>Welcome to the lab</title>
<h1>Welcome to the lab</h1>
<p>Served by web1.</p>

$ curl -s http://web2/
<!doctype html>
<title>Welcome to the lab</title>
<h1>Welcome to the lab</h1>
<p>Served by web2.</p>
```

The same page on both servers, each saying who served it.

14.4. Changing the site

Content changes are changes to vars or templates. To see a new title on one server before writing it into `vars.rb`, plan with `--var`:

```
$ sadame plan site.rb -g web --limit web2 --var 'site_title=Hello, readers'
[web2] sadame 0.5.1 plan on web2 (Debian GNU/Linux 12 (bookworm))
[web2]
[web2] ~ file[/var/www/lab-site/index.html]
[web2]     --- /var/www/lab-site/index.html
[web2]     +++ /var/www/lab-site/index.html (desired)
[web2]     @@ -1,4 +1,4 @@
[web2]      <!doctype html>
[web2]     -<title>Welcome to the lab</title>
[web2]     -<h1>Welcome to the lab</h1>
[web2]     +<title>Hello, readers</title>
[web2]     +<h1>Hello, readers</h1>
[web2]      <p>Served by web2.</p>
[web2]
[web2] Plan: 0 to create, 1 to update, 0 to remove, 0 to run, 6 unchanged.

Hosts
  web2  ok

plan on 1 host: 1 ok. Finished in 0.23s.
```

Only the page would change. The configuration did not, so no reload would be notified: nginx does not need one to serve a new file.

14.5. Going further

A real site needs HTTPS. The usual way is `certbot` with nginx's plugin, which edits the site configuration itself. That collides with a recipe that owns the same file: every run would undo certbot's changes. Two approaches work well:

- Let certbot only fetch certificates (`certbot certonly --webroot`) from an `exec` with a `creates:` guard on the certificate's path, and write the `listen 443 ssl` block yourself in the template. The recipe stays the owner of the configuration.
- Put a proxy that handles certificates on its own, such as Caddy, in front, and keep this recipe as it is.

Either way, the decision is written down in the recipe, where the next person can read it.

What you learned

- Debian's nginx uses `sites-available/` and links in `sites-enabled/`; `link` resources switch sites on and off.
- Several delayed notifications of the same action run once, at the end.
- Reloads finish in the background; a moment later, the new configuration serves.
- Vars hold what is the same everywhere, `locals:` what the recipe computes.

Try it

1. Add a second page, `about.html`, from a template, and link to it from `index.html.erb`.

2. Change `site_name` to `lab-site-2` with `--var` and read the plan. What happens to the old site's link? (Hint: nothing; sadame removes only what a recipe declares `absent`.)
3. Edit `/var/www/lab-site/index.html` on web2 by hand, then run `sadame drift site.rb -g web`.

Chapter 15. Project: A Database Server

The second project sets up PostgreSQL on db1, the Rocky Linux server, for an application that will run on the web servers. It is a more demanding recipe than the website: the database has to be initialized exactly once, tuned to the machine, opened to the right network, and given a role and a database, with a password that must not end up in the repository or in the output.

```
$ cd ~/code/15-database
```

15.1. The plan for the recipe

On Rocky Linux 9, PostgreSQL comes in the `postgresql-server` package. Installing it does not create a database cluster: `postgresql-setup --initdb` does that, once, and writes `PG_VERSION` into the data directory. After that:

- PostgreSQL's settings are in `postgresql.conf`, a long file the package owns. Rather than manage the whole file, the recipe adds one line, `include_dir = 'conf.d'`, and writes its own settings in a small file in that directory.
- Who may connect from where is in `pg_hba.conf`. One line allows password logins from the server's own network.
- The application's role and database are created with SQL, by `exec` resources with guards that ask PostgreSQL whether they exist.

15.2. The recipe

code/15-database/roles/db.rb

```
# PostgreSQL on Rocky Linux: install it, create the cluster once, tune it
# from the machine's facts, let the other lab machines connect, and create
# the application's role and database.
data = "/var/lib/pgsql/data"
psql = "runuser -u postgres -- psql -tA"

package "postgresql-server"

# The cluster is created once: PG_VERSION exists from then on.
exec "create the database cluster",
     command: "postgresql-setup --initdb",
     creates: "#{data}/PG_VERSION"

# Settings in a file of our own, read through include_dir.
directory "#{data}/conf.d", owner: "postgres", group: "postgres", mode: "0700"
file "#{data}/conf.d/tuning.conf",
     template: "tuning.conf.erb",
     owner: "postgres",
     group: "postgres",
     mode: "0600",
     notifies: [:restart, "service[postgresql]"]
line "include_dir = 'conf.d'",
     path: "#{data}/postgresql.conf",
     match: /^#?\s*include_dir\s*=/,
     notifies: [:restart, "service[postgresql]"]

# Password logins from the machines on the server's own network.
line "host all all samenet scram-sha-256",
     path: "#{data}/pg_hba.conf",
     notifies: [:reload, "service[postgresql]"]
```

```
service "postgresql", enable: true, state: :running

# The role and the database, each created once. The password is handed to
# psql through the environment, so plan and apply never print it, and
# stored as SCRAM, which the pg_hba.conf line above asks for. (PostgreSQL 13,
# Rocky 9's version, would otherwise store it as MD5, until the restart at
# the end of this run reads tuning.conf.)
user_exists = "#{psql} -c \"SELECT 1 FROM pg_roles WHERE rolname = '#{vars[:db_user]}'\" | grep -q 1"
exec "create the #{vars[:db_user]} role",
     command: "echo \"SET password_encryption = 'scram-sha-256'; " \
              "CREATE ROLE #{vars[:db_user]} LOGIN PASSWORD :'password';\" | " \
              "runuser -u postgres -- psql -v password=\"$DB_PASSWORD\"",
     env: { DB_PASSWORD: vars[:db_password] },
     not_if: user_exists

db_exists = "#{psql} -c \"SELECT 1 FROM pg_database WHERE datname = '#{vars[:db_name]}'\" | grep -q 1"
exec "create the #{vars[:db_name]} database",
     command: "runuser -u postgres -- createdb --owner #{vars[:db_user]} #{vars[:db_name]}",
     not_if: db_exists
```

Step by step:

`exec "create the database cluster"`:: `creates:` points at `PG_VERSION`. Once the cluster exists, this command never runs again, which matters: running `initdb` on a live cluster would fail at best.

`line "include_dir = 'conf.d'"`:: The stock `postgresql.conf` contains a commented-out `#include_dir = '…'`. The `match:` pattern finds that line, commented or not, and replaces it, instead of adding a second `include_dir` at the end.

`file "…/conf.d/tuning.conf"`

Owned by `postgres` with mode `0600`, like the rest of the data directory. Any change notifies a restart, because PostgreSQL reads most of these settings only at startup.

`line "host all all samenet scram-sha-256"`

Without `match:`, `line` adds the line at the end if it is missing. `samenet` means "any address in a subnet this server is directly connected to", the lab's network here. Changes to `pg_hba.conf` only need a reload.

The role and database `exec` resources use `not_if:` with a query: `psql -tA` prints a bare `1` when the row exists, and `grep -q 1` turns that into an exit status.

15.3. Tuning from facts

code/15-database/roles/templates/tuning.conf.erb

```
# Managed by sadame, from the machine's facts.
listen_addresses = '*'
password_encryption = scram-sha-256
max_connections = <%= vars[:max_connections] %>
shared_buffers = <%= facts[:memory_mb] / 4 %>MB
effective_cache_size = <%= facts[:memory_mb] * 3 / 4 %>MB
```

PostgreSQL's usual starting points are a quarter of the memory for `shared_buffers` and three quarters for `effective_cache_size`. The template computes both from `facts[:memory_mb]`, so a bigger server gets bigger numbers without anyone editing the recipe.

15.4. Keeping the password out

code/15-database/vars.rb

```
# The password comes from the environment of whoever runs sadame, so it is
# not written in the repository. The lab falls back to a lab-only value.
{
  db_name: "inventory",
  db_user: "app",
  db_password: ENV.fetch("APP_DB_PASSWORD", "lab-only-password"),
  max_connections: 50
}
```

Three measures keep the password where it belongs:

1. **Not in the repository.** `vars.rb` reads it from the environment of whoever runs sadame: `APP_DB_PASSWORD=… sadame apply …`. `vars.rb` runs on your computer, not the server; the lab falls back to a throwaway value so the book's commands work as printed.
2. **Not in the recipe's command.** The `exec` hands it over in an environment variable, `env: { DB_PASSWORD: … }`; the shell passes it to `psql` as the variable `password`, and the SQL uses it as `:'password'`, which psql quotes safely whatever characters it contains.
3. **Not in the output.** The command sadame prints shows `$DB_PASSWORD`, never the value, as you will see.

While `psql` runs, for a fraction of a second, its arguments contain the value, and a program's arguments are visible to every user of a Linux machine. On a database server that only administrators log in to, that is acceptable; where it is not, have `psql` read the value itself with `\set password `printenv DB_PASSWORD``.

For a file whose content is secret, such as a configuration file that contains a password, add `sensitive: true`: sadame then prints that the content changed but not the diff.



During a run, the target's merged vars are written to a file in the run's directory under `~/.sadame/run/`, readable only by the user sadame logs in as, and removed when the run ends. For a production database, use a password manager or your cloud provider's secret store to fill `APP_DB_PASSWORD` on the machine that runs sadame.

15.5. Running it

`site.rb` gives db1 the `db` role and the web servers the PostgreSQL client. Plan the database server:

```
$ sadame plan site.rb -g db
[db1] sadame 0.5.1 plan on db1 (Rocky Linux 9.8 (Blue Onyx))
[db1]
[db1] + package[postgresql-server]
[db1]     state: absent -> installed
[db1]
[db1] ! exec[create the database cluster]
[db1]     command: postgresql-setup --initdb
[db1]     (/var/lib/pgsql/data/PG_VERSION does not exist)
[db1]
```

```
[db1] + directory[/var/lib/pgsql/data/conf.d]
[db1]     state: absent -> present
[db1]     mode: 0700
[db1]     owner: postgres
[db1]     group: postgres
[db1]
[db1] + file[/var/lib/pgsql/data/conf.d/tuning.conf]
[db1]     state: absent -> present
[db1]     mode: 0600
[db1]     owner: postgres
[db1]     group: postgres
[db1]     --- /dev/null
[db1]     +++ /var/lib/pgsql/data/conf.d/tuning.conf (desired)
[db1]     @@ -0,0 +1,6 @@
[db1]     +# Managed by sadame, from the machine's facts.
[db1]     +listen_addresses = '*'
[db1]     +password_encryption = scram-sha-256
[db1]     +max_connections = 50
[db1]     +shared_buffers = 2000MB
[db1]     +effective_cache_size = 6002MB
[db1]     notifies: restart service[postgresql] (delayed)
[db1]
[db1] ~ line[include_dir = 'conf.d']
[db1]     --- /dev/null
[db1]     +++ /var/lib/pgsql/data/postgresql.conf (desired)
[db1]     @@ -0,0 +1 @@
[db1]     +include_dir = 'conf.d'
[db1]     (/var/lib/pgsql/data/postgresql.conf does not exist yet)
[db1]     notifies: restart service[postgresql] (delayed)
[db1]
[db1] ~ line[host all all samenet scram-sha-256]
[db1]     --- /dev/null
[db1]     +++ /var/lib/pgsql/data/pg_hba.conf (desired)
[db1]     @@ -0,0 +1 @@
[db1]     +host all all samenet scram-sha-256
[db1]     (/var/lib/pgsql/data/pg_hba.conf does not exist yet)
[db1]     notifies: reload service[postgresql] (delayed)
[db1]
[db1] ~ service[postgresql]
[db1]     enabled: false -> true
[db1]     state: stopped -> running
[db1]
[db1] ! exec[create the app role]
[db1]     command: echo "SET password_encryption = 'scram-sha-256'; CREATE ROLE app LOGIN PASSWORD :'password';"
| runuser -u postgres -- psql -v password="$DB_PASSWORD"
[db1]     (not_if exited non-zero)
[db1]
[db1] ! exec[create the inventory database]
[db1]     command: runuser -u postgres -- createdb --owner app inventory
[db1]     (not_if exited non-zero)
[db1]
[db1] Plan: 3 to create, 3 to update, 0 to remove, 3 to run, 0 unchanged.

Hosts
  db1  ok

plan on 1 host: 1 ok. Finished in 0.23s.
```

Look at the two `line` resources. Their files, `postgresql.conf` and `pg_hba.conf`, do not exist yet: the cluster has not been created. The plan says so, `(… does not exist yet)`, and shows the line as the whole new content. A plan reads the machine as it is now; it cannot know what `postgresql-setup` will write. At apply time, the line resources run after the cluster exists and read the real files.

The `exec` commands show `$DB_PASSWORD`, not a password. Apply to the database and the web servers together:

```
$ sadame apply site.rb -g db,web > apply.log; echo $?
0
```

```
$ grep -E '^\[db1\] (changed|notified)' apply.log
[db1] changed    package[postgresql-server]
[db1] changed    exec[create the database cluster]
[db1] changed    directory[/var/lib/pgsql/data/conf.d]
[db1] changed    file[/var/lib/pgsql/data/conf.d/tuning.conf]
[db1] changed    line[include_dir = 'conf.d']
[db1] changed    line[host all all samenet scram-sha-256]
[db1] changed    service[postgresql]
[db1] changed    exec[create the app role]
[db1] changed    exec[create the inventory database]
[db1] notified   service[postgresql]: restart (delayed, from file[/var/lib/pgsql/data/conf.d/tuning.conf])
[db1] notified   service[postgresql]: reload (delayed, from line[host all all samenet scram-sha-256])

$ tail -n 7 apply.log

Hosts
  db1   ok
  web1  ok
  web2  ok

apply on 3 hosts: 3 ok. Finished in 14.89s.
```

Two notifications: the restart for the tuning file, and the reload for `pg_hba.conf`. Each ran once, at the end. (The `include_dir` line also asked for the restart; it collapsed into the same one.)

15.6. Does it work?

Connect from web1, as the application will:

```
$ ssh web1 "psql postgresql://app:lab-only-password@db1/inventory -c 'SELECT current_user, current_database()'"
 current_user | current_database
--------------+------------------
 app          | inventory
(1 row)
```

The `app` role logs in to its `inventory` database over the network, with a password checked by SCRAM, from another machine. Plan once more:

```
$ sadame plan site.rb -g db
[db1] sadame 0.5.1 plan on db1 (Rocky Linux 9.8 (Blue Onyx))
[db1]
[db1] Plan: 0 to create, 0 to update, 0 to remove, 0 to run, 9 unchanged.

Hosts
  db1  ok

plan on 1 host: 1 ok. Finished in 0.23s.
```

Nine resources, all unchanged, including the three commands: their guards say the work is done.

What you learned

- `creates:` makes one-time setup commands safe to keep in a recipe.
- `line` with `match:` replaces a setting in a file another package owns; without `match:`, it adds a line once.
- Templates can size configuration from `facts`.
- Keep secrets in the environment of the machine running sadame, pass them to commands with `env:`, and mark secret files `sensitive: true`.

- A plan cannot see what earlier resources will create; apply can.

Try it

1. Run the apply again with `APP_DB_PASSWORD=another sadame apply site.rb -g db`. Does the password change? Why not? (Look at the guard.)
2. Add `work_mem = 16MB` to the template and plan. Which notification appears?
3. Create a second database, `reports`, owned by `app`, with another guarded `exec`.

Chapter 16. Project: Deploying an Application

The third project deploys an application: code from a git repository, run as a service by systemd, under its own user. Then it does what makes deployment hard: it upgrades the application to a new version on both web servers, one server at a time, so the service never goes down everywhere at once.

```
$ cd ~/code/16-application
```

16.1. The application

The book's repository contains a small application, `hello-app`: a Python web server, using only Python's standard library, that answers every request with its version and the name of the machine it runs on. Its two releases are git tags in the repository, `hello-app-1.0` and `hello-app-1.1`. Version 1.1 differs from 1.0 by one line, its version number, which is all a deployment exercise needs.

16.2. The recipe

code/16-application/roles/app.rb

```
# hello-app from its git repository, at the release vars[:app_version]
# names, run by systemd as its own user, and restarted whenever the code
# or the unit changes.
home = "/srv/hello-app"

package "python3"
package "git"

user "app", system: true, home: home, shell: "/usr/sbin/nologin"
directory home, owner: "app", group: "app", mode: "0755"

git home,
    repository: "https://github.com/enderkus/sadame-beginner-guide.git",
    revision: "hello-app-#{vars[:app_version]}",
    user: "app"

file "/etc/systemd/system/hello-app.service",
     template: "hello-app.service.erb",
     locals: { home: home },
     mode: "0644",
     notifies: [[:run, "exec[systemctl daemon-reload]", :immediately], [:restart, "service[hello-app]"]]

exec "systemctl daemon-reload", action: :nothing

service "hello-app",
        enable: true,
        state: :running,
        subscribes: [:restart, "git[#{home}]"]
```

code/16-application/vars.rb

```
{
  app_version: "1.0",
  app_port: 8000,
  greeting: "Hello"
}
```

code/16-application/roles/templates/hello-app.service.erb

```
# Managed by sadame.
[Unit]
Description=hello-app <%= vars[:app_version] %>
After=network.target

[Service]
User=app
Environment=PORT=<%= vars[:app_port] %>
Environment=GREETING=<%= vars[:greeting] %>
ExecStart=/usr/bin/python3 <%= locals[:home] %>/app/server.py
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Four ideas carry this recipe.

Its own user. `user "app"` with `system: true` and the shell `/usr/sbin/nologin`: an account that runs the application and nothing else, which nobody can log in as. If the application is compromised, the attacker has `app's rights, not root's.

The code from git. The `git` resource clones a repository into a directory and keeps it at a `revision:`, a branch, a tag or a commit. Here the revision comes from a var, `hello-app-#{vars[:app_version]}`, so a release is one changed var. `user: "app"` runs git as that user, so the files belong to it.

A systemd unit. The template is a service definition: run `server.py` as `app`, with the port and greeting in its environment, restart it if it fails, start it at boot. Its `Description` includes the version, which makes the running release visible in `systemctl status`.

Two kinds of notification. systemd reads unit files into memory, and must be told to read them again, with `systemctl daemon-reload`, after one changes. That has to happen **before** the service is restarted, or systemd restarts the old definition. So the unit file notifies the reload `:immediately`, right after the file is written, and the restart delayed, at the end. The service also `subscribes:` to the git checkout: new code means a restart.

16.3. The first deployment

Plan on web1:

```
$ sadame plan site.rb -g web --limit web1
[web1] sadame 0.5.1 plan on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] + package[python3]
[web1]     state: absent -> installed
[web1]
[web1] + package[git]
[web1]     state: absent -> installed
[web1]
[web1] + user[app]
[web1]     state: absent -> present
[web1]     home: /srv/hello-app
[web1]     shell: /usr/sbin/nologin
[web1]
[web1] + directory[/srv/hello-app]
[web1]     state: absent -> present
[web1]     mode: 0755
[web1]     owner: app
```

```
[web1]     group: app
[web1]
[web1] + git[/srv/hello-app]
[web1]     state: absent -> present
[web1]     repository: https://github.com/enderkus/sadame-beginner-guide.git
[web1]     revision: hello-app-1.0
[web1]     notifies: restart service[hello-app] (delayed)
[web1]
[web1] + file[/etc/systemd/system/hello-app.service]
[web1]     state: absent -> present
[web1]     mode: 0644
[web1]     --- /dev/null
[web1]     +++ /etc/systemd/system/hello-app.service (desired)
[web1]     @@ -0,0 +1,14 @@
[web1]     +# Managed by sadame.
[web1]     +[Unit]
[web1]     +Description=hello-app 1.0
[web1]     +After=network.target
[web1]     +
[web1]     +[Service]
[web1]     +User=app
[web1]     +Environment=PORT=8000
[web1]     +Environment=GREETING=Hello
[web1]     +ExecStart=/usr/bin/python3 /srv/hello-app/app/server.py
[web1]     +Restart=on-failure
[web1]     +
[web1]     +[Install]
[web1]     +WantedBy=multi-user.target
[web1]     notifies: run exec[systemctl daemon-reload] (immediately)
[web1]     notifies: restart service[hello-app] (delayed)
[web1]
[web1] ~ service[hello-app]
[web1]     enabled: false -> true
[web1]     state: stopped -> running
[web1]
[web1] Plan: 6 to create, 1 to update, 0 to remove, 0 to run, 1 unchanged.

Hosts
  web1  ok

plan on 1 host: 1 ok. Finished in 0.22s.
```

The plan lists both notifications of the unit file with their timing, `(immediately)` and `(delayed)`. The one unchanged resource is the `exec` with `action: :nothing`, which never runs on its own. Apply on both servers:

```
$ sadame apply site.rb -g web > apply.log; echo $?
0

$ tail -n 6 apply.log

Hosts
  web1  ok
  web2  ok

apply on 2 hosts: 2 ok. Finished in 16.94s.
```

Most of that time went into installing git and Python. The service needs a moment to start listening; then:

```
$ curl -s http://web1:8000/
Hello from hello-app 1.0 on web1

$ curl -s http://web2:8000/
Hello from hello-app 1.0 on web2

$ ssh web1 sudo systemctl is-active hello-app
active
```

16.4. Releasing version 1.1

A new release is a change of one var. Change it in `vars.rb` and plan on web1:

```
$ sed -i 's/app_version: "1.0"/app_version: "1.1"/' vars.rb

$ sadame plan site.rb -g web --limit web1
[web1] sadame 0.5.1 plan on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] ~ git[/srv/hello-app]
[web1]     revision: d2d0295ed157c5331df5c06d9372d9c4c02543eb -> 688f6d1316ea1a185f2a1a4c7c4a7f1b052d4423
[web1]     notifies: restart service[hello-app] (delayed)
[web1]
[web1] ~ file[/etc/systemd/system/hello-app.service]
[web1]     --- /etc/systemd/system/hello-app.service
[web1]     +++ /etc/systemd/system/hello-app.service (desired)
[web1]     @@ -1,6 +1,6 @@
[web1]      # Managed by sadame.
[web1]      [Unit]
[web1]     -Description=hello-app 1.0
[web1]     +Description=hello-app 1.1
[web1]      After=network.target
[web1]
[web1]      [Service]
[web1]     notifies: run exec[systemctl daemon-reload] (immediately)
[web1]     notifies: restart service[hello-app] (delayed)
[web1]
[web1] Plan: 0 to create, 2 to update, 0 to remove, 0 to run, 6 unchanged.

Hosts
  web1  ok

plan on 1 host: 1 ok. Finished in 0.84s.
```

The plan shows the revision as commit IDs: the git resource resolved both tags to the commits they point to. The unit file changes because its description contains the version.

16.5. One server at a time

By default sadame runs on all selected servers at once. For an upgrade, that means both web servers restart the application at the same moment. `--serial 1` runs them in batches of one, and stops before the next batch if a host fails:

```
$ sadame apply site.rb -g web --serial 1 > upgrade.log; echo $?
0

$ sed -n '/^Batch 1/,/^\[web1\] Summary/p' upgrade.log
Batch 1 of 2: web1
[web1] sadame 0.5.1 apply on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] unchanged  package[python3]
[web1] unchanged  package[git]
[web1] unchanged  user[app]
[web1] unchanged  directory[/srv/hello-app]
[web1] changed    git[/srv/hello-app]
[web1]            revision: d2d0295ed157c5331df5c06d9372d9c4c02543eb -> 688f6d1316ea1a185f2a1a4c7c4a7f1b052d4423
[web1] changed    file[/etc/systemd/system/hello-app.service]
[web1]            --- /etc/systemd/system/hello-app.service
[web1]            +++ /etc/systemd/system/hello-app.service (desired)
[web1]            @@ -1,6 +1,6 @@
[web1]             # Managed by sadame.
[web1]             [Unit]
[web1]            -Description=hello-app 1.0
[web1]            +Description=hello-app 1.1
[web1]             After=network.target
[web1]
[web1]             [Service]
```

```
[web1] notified   exec[systemctl daemon-reload]: run (immediately, from file[/etc/systemd/system/hello-
app.service])
[web1] unchanged  exec[systemctl daemon-reload]
[web1] unchanged  service[hello-app]
[web1] notified   service[hello-app]: restart (delayed, from git[/srv/hello-app])
[web1]
[web1] Summary

$ grep -E '^Batch 2|^\[web2\] (changed|notified) ' upgrade.log
Batch 2 of 2: web2
[web2] changed    git[/srv/hello-app]
[web2] changed    file[/etc/systemd/system/hello-app.service]
[web2] notified   exec[systemctl daemon-reload]: run (immediately, from file[/etc/systemd/system/hello-
app.service])
[web2] notified   service[hello-app]: restart (delayed, from git[/srv/hello-app])

$ tail -n 6 upgrade.log

Hosts
  web1  ok
  web2  ok

apply on 2 hosts: 2 ok. Finished in 3.05s.
```

Follow web1's lines from top to bottom:

1. The git checkout moves to the new commit.
2. The unit file changes, and its **immediate** notification runs `systemctl daemon-reload` right away.
3. The `exec` itself is reported `unchanged` in its place: it has `action: :nothing` and only ran through the notification.
4. At the end, the delayed restart runs once, though both the git checkout and the unit file asked for it.

Only then does web2 start, with the same changes. If web1 had failed, sadame would have stopped there, with web2 still running version 1.0 and untouched. A moment later, both answer with the new version:

```
$ curl -s http://web1:8000/
Hello from hello-app 1.1 on web1

$ curl -s http://web2:8000/
Hello from hello-app 1.1 on web2
```

`--serial` also takes a percentage, `--serial 25%`, for larger groups: four batches, each a quarter of the hosts. Combined with a load balancer that only sends traffic to healthy servers, that is a rolling upgrade with no downtime.



To roll back, set `app_version` back to `"1.0"` and apply again. A deployment that is a value in a recipe is also a rollback that is a value in a recipe, and `git log` on your recipes tells you who released what, and when.

What you learned

- `git` keeps a checkout at a revision; a var in the revision turns a release into a one-line change.
- Applications run as their own system users, under a systemd unit the recipe writes.

- After a unit file changes, `daemon-reload` must run before the restart: notify it `:immediately`.
- `--serial 1` and `--serial 25%` upgrade a group in batches and stop at the first failure.

Try it

1. Change `greeting` to `Merhaba` and apply with `--serial 1`. Which resources change, and why does the service restart?
2. Roll back to version 1.0 on web2 only, with `--limit web2 --var app_version=1.0`.
3. Run `ssh web1 sudo systemctl status hello-app` and find the version in the output.

Chapter 17. Project: The Whole Fleet

The last project puts the three previous ones together, adds a role every server should have, and builds the entire lab from new, empty servers with one command. This recipe is the one to copy when you start managing your own servers: one entry point, one role per job, all values in one place, and every role runnable on its own.

```
$ cd ~/code/17-fleet
```

17.1. The shape of it

code/17-fleet/site.rb

```
# The whole lab in one recipe. Every server gets the base and the
# hardening; the web servers serve the site and run hello-app behind it;
# the database server runs PostgreSQL. Each role is tagged with its name,
# so --tags runs any of them on its own.
include_recipe "roles/base", tags: [:base]
include_recipe "roles/hardening", tags: [:hardening]

if host_groups.include?(:web)
  include_recipe "roles/web", tags: [:web]
  include_recipe "roles/app", tags: [:app]
  include_recipe "roles/db_client", tags: [:db_client]
end

include_recipe "roles/db", tags: [:db] if host_groups.include?(:db)
```

Every server gets `base` and `hardening`. A web server also serves the website, runs hello-app behind it, and has the PostgreSQL client. The database server runs PostgreSQL. The roles are the ones from chapters 13 to 16, with two changes: `base` writes the login message from chapter 11, and the site's nginx configuration sends `/app/` to hello-app:

code/17-fleet/roles/templates/site.conf.erb

```
# Managed by sadame.
server {
    listen 80 default_server;
    server_name <%= vars[:server_name] %>;
    root <%= locals[:root] %>;
    index index.html;

    # hello-app, which runs on this server.
    location /app/ {
        proxy_pass http://127.0.0.1:<%= vars[:app_port] %>/;
    }
}
```

All values live in one `vars.rb`, grouped by the role that reads them:

code/17-fleet/vars.rb

```
# Everything the roles read, in one place. The database password comes from
# the environment of whoever runs sadame; the lab falls back to a lab-only
# value.
{
  contact: "ops@example.com",
  timezone: "Europe/Istanbul",

  site_name: "lab-site",
  site_title: "Welcome to the lab",
```

```
  server_name: "_",

  app_version: "1.1",
  app_port: 8000,
  greeting: "Hello",

  db_name: "inventory",
  db_user: "app",
  db_password: ENV.fetch("APP_DB_PASSWORD", "lab-only-password"),
  max_connections: 50
}
```

When somebody asks "which version runs in production?" or "how many database connections do we allow?", this file answers.

17.2. A new role: hardening

code/17-fleet/roles/hardening.rb

```
# SSH that refuses root logins and gives up after three wrong attempts, and
# security updates that install themselves. The SSH service and the update
# tool have different names on Debian and Rocky Linux.
ssh_service = platform.family == "debian" ? "ssh" : "sshd"

# sshd reads the files in sshd_config.d/ before the rest of sshd_config,
# in name order, and keeps the first value it finds for each setting. A
# file named 00-... comes first, so these settings win over any a cloud
# image or installer put there, such as Rocky's 01-permitrootlogin.conf.
ssh_settings = { "PermitRootLogin" => "no", "MaxAuthTries" => "3" }

file "/etc/ssh/sshd_config.d/00-hardening.conf",
     content: ssh_settings.map { |option, value| "#{option} #{value}\n" }.join,
     mode: "0644",
     verify: "sshd -t -f %{path}",
     notifies: [:restart, "service[#{ssh_service}]"]

service ssh_service, enable: true, state: :running

if platform.family == "debian"
  package "unattended-upgrades"
  file "/etc/apt/apt.conf.d/20auto-upgrades",
       content: "APT::Periodic::Update-Package-Lists \"1\";\nAPT::Periodic::Unattended-Upgrade \"1\";\n",
       mode: "0644"
else
  package "dnf-automatic"
  service "dnf-automatic-install.timer", enable: true, state: :running
end
```

Two measures most servers on the internet should have:

- SSH refuses logins as root and gives up after three wrong attempts.
- Security updates install themselves: `unattended-upgrades` on Debian, `dnf-automatic` with its install timer on Rocky Linux.

The SSH settings are not lines in `sshd_config`, though `line` could write them there, but a file of their own in `sshd_config.d/`. The reason is a lesson this book learned the hard way, testing these recipes on real virtual machines. Modern `sshd_config` files begin with `Include /etc/ssh/sshd_config.d/*.conf`, and for each setting sshd keeps the *first* value it reads. Rocky Linux's installer puts a `01-permitrootlogin.conf` there, and cloud images often add their own files. A `PermitRootLogin no` line further down in `sshd_config` is then silently ignored: the recipe reports

success, `drift` is clean, and root can still log in. A file named `00-hardening.conf` is read before all of them, so its values win.



Check what a program actually uses, not only what its files say. For sshd, `sudo sshd -T` prints the effective configuration: `sudo sshd -T | grep -i permitrootlogin`.

The file's content is built from a Ruby Hash with `map` and `join`: add a setting to the Hash, and it is in the file.

The `verify:` option is what makes changing SSH's configuration safe. Before sadame replaces the file, it writes the new version next to it and runs `sshd -t -f` on it. If sshd rejects it, the file is left as it was and the resource fails. A broken SSH configuration on a server you reach only by SSH is the classic way to lock yourself out; `verify:` stops it before it happens. Chapter 19 shows it at work.

17.3. Check before you build

A recipe this size deserves a check before it meets any server:

```
$ sadame validate site.rb -g all
sadame 0.5.1 validate site.rb, as on Debian GNU/Linux 12 (bookworm)

  web1  ok     22 resources
  web2  ok     22 resources
  db1   ok     15 resources

validate on 3 hosts: 3 ok.
```

`validate` evaluated the recipe for every host, with its groups and vars, and counted the resources each would get, without connecting to any. Chapter 19 shows what it catches.

17.4. From empty servers to a fleet

The lab's servers have been reset: new containers with nothing but SSH and the `deploy` user, as you would get from a cloud provider. Build everything, one server at a time:

```
$ sadame apply site.rb -g all --serial 1 > apply.log; echo $?
0

$ grep '^Batch' apply.log
Batch 1 of 3: web1
Batch 2 of 3: web2
Batch 3 of 3: db1

$ sed -n '/^\[db1\] Summary/,/^\[db1\] Finished/p' apply.log
[db1] Summary
[db1]   changed     14  file[/etc/motd]
[db1]                   link[/etc/localtime]
[db1]                   file[/etc/ssh/sshd_config.d/00-hardening.conf]
[db1]                   package[dnf-automatic]
[db1]                   service[dnf-automatic-install.timer]
[db1]                   package[postgresql-server]
[db1]                   exec[create the database cluster]
[db1]                   directory[/var/lib/pgsql/data/conf.d]
[db1]                   file[/var/lib/pgsql/data/conf.d/tuning.conf]
[db1]                   line[include_dir = 'conf.d']
```

```
[db1]                   line[host all all samenet scram-sha-256]
[db1]                   service[postgresql]
[db1]                   exec[create the app role]
[db1]                   exec[create the inventory database]
[db1]   unchanged    1
[db1]   failed       0
[db1]   skipped      0
[db1]   notified     3  service[sshd]: restart
[db1]                   service[postgresql]: restart
[db1]                   service[postgresql]: reload
[db1]
[db1] Finished in 33.90s.

$ tail -n 6 apply.log
Hosts
  web1  ok
  web2  ok
  db1   ok

apply on 3 hosts: 3 ok. Finished in 78.35s.
```

Three batches, one per server, each finished before the next started. db1's summary lists fourteen changes in the order of the recipe: the base, the hardening, then the database role. The one unchanged resource is `service[sshd]`, which was running already. At the end, three notifications: `sshd` restarted for its new settings, PostgreSQL restarted for its tuning and reloaded for `pg_hba.conf`. Under a minute and a half for three servers from nothing, most of it spent installing packages.

Does it work?

```
$ curl -s http://web1/app/
Hello from hello-app 1.1 on web1

$ curl -s http://web2/ | grep h1
<h1>Welcome to the lab</h1>

$ sadame drift site.rb -g all | tail -n 6
Hosts
  web1  ok
  web2  ok
  db1   ok

drift on 3 hosts: 3 ok. Finished in 0.90s.
```

The site answers on both web servers, hello-app answers behind nginx at `/app/`, and drift finds every server exactly as the recipe describes.

17.5. Running one role

Tags from `site.rb` work on the whole fleet. Check only the hardening, everywhere:

```
$ sadame plan site.rb -g all --tags hardening
[web1] sadame 0.5.1 plan on web1 (Debian GNU/Linux 12 (bookworm))
[web1] Only resources tagged hardening: 4 of 22.
[web1]
[web2] sadame 0.5.1 plan on web2 (Debian GNU/Linux 12 (bookworm))
[web2] Only resources tagged hardening: 4 of 22.
[web2]
[web2] Plan: 0 to create, 0 to update, 0 to remove, 0 to run, 4 unchanged.
[web1] Plan: 0 to create, 0 to update, 0 to remove, 0 to run, 4 unchanged.
[db1] sadame 0.5.1 plan on db1 (Rocky Linux 9.8 (Blue Onyx))
[db1] Only resources tagged hardening: 4 of 15.
[db1]
[db1] Plan: 0 to create, 0 to update, 0 to remove, 0 to run, 4 unchanged.
```

```
Hosts
  web1  ok
  web2  ok
  db1   ok

plan on 3 hosts: 3 ok. Finished in 0.26s.
```

Four resources per server, all as they should be. This is the command to run after reading a security advisory about SSH: it reads only what the advisory is about, on every server, in a fraction of a second.

17.6. Where to go from here

This recipe is a template for real infrastructure. To adapt it:

1. Replace the hosts in `hosts.rb` with your servers, or a dynamic inventory from your cloud provider (chapter 11).
2. Put the repository under git, if it is not already, and review changes as pull requests: a plan pasted into a pull request is a precise description of what the change will do.
3. Add a scheduled drift check, chapter 18.
4. Add a role at a time. A role that only adds a package and a configuration file is a good first one.

What you learned

- One `site.rb`, one role per job, one `vars.rb`: a fleet's recipe reads like a description of the fleet.
- `verify:` checks a file's new content with the program that will read it before replacing it.
- `validate` checks the whole fleet's recipe without touching a server.
- The same recipe builds new servers from nothing and keeps existing ones as described.

Try it

1. Put web2 into the `db` group as well, run `sadame validate site.rb -g all`, and compare its resource count with web1's.
2. Add `PasswordAuthentication no` to the hardening Hash and plan with `--tags hardening`.
3. Roll the whole fleet back to hello-app 1.0, one server at a time: `sadame apply site.rb -g web --tags app --serial 1 --var app_version=1.0`. Then apply without the `--var` to return to 1.1.

Part V: Day Two

Chapter 18. Drift Detection and CI

Part V is about the time after the first deployment: day two, and every day after it. Servers do not stay as you left them. Someone fixes an incident at three in the morning by editing a file and forgets to tell the recipe. A package upgrade rewrites a configuration. A service stops and nobody notices for a week. Each of these is **drift**, and this chapter turns `sadame drift` into a daily check that tells you about it.

```
$ cd ~/code/17-fleet
```

18.1. Three small accidents

Make three changes by hand, the kind that happen during a busy week: a "temporary" page title on web2, hello-app stopped on web1, and SSH's `MaxAuthTries` raised on db1:

```
$ ssh web2 "sudo sed -i 's/Welcome to the lab/Under new management/' /var/www/lab-site/index.html"

$ ssh web1 sudo systemctl stop hello-app

$ ssh db1 "sudo sed -i 's/^MaxAuthTries 3/MaxAuthTries 10/' /etc/ssh/sshd_config.d/00-hardening.conf"
```

Now ask the fleet:

```
$ sadame drift site.rb -g all
[web1] sadame 0.5.1 drift on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web2] sadame 0.5.1 drift on web2 (Debian GNU/Linux 12 (bookworm))
[web2]
[web2] ~ file[/var/www/lab-site/index.html]
[web2]     --- /var/www/lab-site/index.html
[web2]     +++ /var/www/lab-site/index.html (desired)
[web2]     @@ -1,4 +1,4 @@
[web2]      <!doctype html>
[web2]     -<title>Under new management</title>
[web2]     -<h1>Under new management</h1>
[web2]     +<title>Welcome to the lab</title>
[web2]     +<h1>Welcome to the lab</h1>
[web2]      <p>Served by web2.</p>
[web2]
[db1] sadame 0.5.1 drift on db1 (Rocky Linux 9.8 (Blue Onyx))
[db1]
[db1] ~ file[/etc/ssh/sshd_config.d/00-hardening.conf]
[db1]     --- /etc/ssh/sshd_config.d/00-hardening.conf
[db1]     +++ /etc/ssh/sshd_config.d/00-hardening.conf (desired)
[db1]     @@ -1,2 +1,2 @@
[db1]      PermitRootLogin no
[db1]     -MaxAuthTries 10
[db1]     +MaxAuthTries 3
[db1]     notifies: restart service[sshd] (delayed)
[db1]
[db1] Drift: 1 of 15 resources differs from the recipe.
[web1] ~ service[hello-app]
[web1]     state: stopped -> running
[web1]
[web1] Drift: 1 of 22 resources differs from the recipe.
[web2] Drift: 1 of 22 resources differs from the recipe.

Hosts
  web1  drift
```

```
  web2  drift
  db1   drift

drift on 3 hosts: 3 drift. Finished in 0.92s.

$ echo $?
1
```

Each change found, on the right server, with what differs: the page's diff, the stopped service, the SSH setting in its file. And the exit status, `1`, which is what makes drift useful to machines as well as people:

Status	Meaning
`0`	Every server matches the recipe.
`1`	At least one server differs from the recipe.
`2`	A server could not be checked: unreachable, a recipe error, or a state sadame could not read.

`drift` changes nothing. It reads the same state `plan` reads, so it is safe to run as often as you like, on production, by a robot.

18.2. Reports for people

With `--json`, drift writes one JSON document with every host's result. `sadame report` turns that into Markdown, for a pull request, an issue or a CI summary:

````
$ sadame drift site.rb -g all --json > drift.json

$ sadame report drift.json
### sadame drift on 3 hosts: 3 drift

| Host | Result | Resources |
|---|---|---|
| web1 | **drift** | 1 of 22 differ |
| web2 | **drift** | 1 of 22 differ |
| db1 | **drift** | 1 of 15 differ |

<details>
<summary>web1: 1 of 22 differ</summary>

**~ service[hello-app]**

- state: stopped -> running

</details>

<details>
<summary>web2: 1 of 22 differ</summary>

**~ file[/var/www/lab-site/index.html]**

```diff
--- current
+++ desired
@@ -1,4 +1,4 @@
 <!doctype html>
-<title>Under new management</title>
-<h1>Under new management</h1>
+<title>Welcome to the lab</title>
+<h1>Welcome to the lab</h1>
 <p>Served by web2.</p>
```
````

```
</details>

<details>
<summary>db1: 1 of 15 differ</summary>

**~ file[/etc/ssh/sshd_config.d/00-hardening.conf]**

```diff
--- current
+++ desired
@@ -1,2 +1,2 @@
 PermitRootLogin no
-MaxAuthTries 10
+MaxAuthTries 3
```

</details>
```

On GitHub, the `<details>` sections fold, so the summary shows one line per host and the diffs open on a click.

CI systems such as GitLab and Jenkins understand JUnit XML, the format of test results. `--format junit` writes each resource as a test case and each drifted resource as a failure, so the CI shows drift where it shows failed tests:

```
$ sadame report drift.json --format junit | head -n 22
<?xml version="1.0" encoding="UTF-8"?>
<testsuites name="sadame drift" tests="59" failures="3" errors="0">
  <testsuite name="web1" tests="22" failures="1" errors="0">
    <testcase classname="web1" name="file[/etc/motd]"/>
    <testcase classname="web1" name="link[/etc/localtime]"/>
    <testcase classname="web1" name="file[/etc/ssh/sshd_config.d/00-hardening.conf]"/>
    <testcase classname="web1" name="service[ssh]"/>
    <testcase classname="web1" name="package[unattended-upgrades]"/>
    <testcase classname="web1" name="file[/etc/apt/apt.conf.d/20auto-upgrades]"/>
    <testcase classname="web1" name="package[nginx]"/>
    <testcase classname="web1" name="directory[/var/www/lab-site]"/>
    <testcase classname="web1" name="file[/var/www/lab-site/index.html]"/>
    <testcase classname="web1" name="file[/etc/nginx/sites-available/lab-site]"/>
    <testcase classname="web1" name="link[/etc/nginx/sites-enabled/lab-site]"/>
    <testcase classname="web1" name="link[/etc/nginx/sites-enabled/default]"/>
    <testcase classname="web1" name="service[nginx]"/>
    <testcase classname="web1" name="package[python3]"/>
    <testcase classname="web1" name="package[git]"/>
    <testcase classname="web1" name="user[app]"/>
    <testcase classname="web1" name="directory[/srv/hello-app]"/>
    <testcase classname="web1" name="git[/srv/hello-app]"/>
    <testcase classname="web1" name="file[/etc/systemd/system/hello-app.service]"/>
```

18.3. Repairing

Repairing drift is an apply:

```
$ sadame apply site.rb -g all > repair.log; echo $?
0

$ grep -E '^\[[a-z0-9]+\] (changed|notified) ' repair.log
[web2] changed    file[/var/www/lab-site/index.html]
[db1] changed    file[/etc/ssh/sshd_config.d/00-hardening.conf]
[db1] notified   service[sshd]: restart (delayed, from file[/etc/ssh/sshd_config.d/00-hardening.conf])
[web1] changed    service[hello-app]

$ sadame drift site.rb -g all | tail -n 6
Hosts
  web1  ok
  web2  ok
```

```
  db1   ok

drift on 3 hosts: 3 ok. Finished in 0.93s.
```

Three changes, one per accident, and the SSH restart that its setting needs. The fleet matches again.



Not every drift should be undone. If the 3 a.m. edit was right, the fix is to change the recipe, not the server: put the edit into the template, plan, and see the drift disappear without anything changing. Either way, after a drift report, the recipe and the servers agree again, and the recipe is still the truth.

18.4. A daily check in CI

A drift check is only useful if it runs when nobody remembers to run it. Any CI system with a schedule can do it. For GitHub Actions:

code/18-drift-and-ci/drift.yml

```yaml
# A daily drift check for the fleet of chapter 17, for GitHub Actions: copy
# it to .github/workflows/drift.yml in the repository that holds your
# recipes. `sadame init --ci github` writes the same job for a new project.
#
# It needs repository secrets SSH_PRIVATE_KEY (a key the hosts.rb users
# accept) and SSH_KNOWN_HOSTS (the output of `ssh-keyscan` for every
# server), and passwordless sudo on the servers.
name: Drift

on:
  schedule:
    - cron: "0 6 * * *"
  workflow_dispatch:

jobs:
  drift:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7

      - name: Install sadame
        run: curl -fsSL https://raw.githubusercontent.com/enderkus/sadame/main/install.sh | sh

      - name: Set up SSH
        run: |
          install -d -m 0700 ~/.ssh
          printf '%s\n' "${{ secrets.SSH_PRIVATE_KEY }}" > ~/.ssh/id_ed25519
          chmod 0600 ~/.ssh/id_ed25519
          printf '%s\n' "${{ secrets.SSH_KNOWN_HOSTS }}" > ~/.ssh/known_hosts

      # Exit status: 0 no drift, 1 drift on a server, 2 a server could not be
      # checked. Anything but 0 fails this step and the workflow.
      - name: Check for drift
        run: sadame drift 17-fleet/site.rb -g all --json > drift.json

      - name: Summarize
        if: always()
        run: sadame report drift.json >> "$GITHUB_STEP_SUMMARY"

      - name: Keep the drift report
        if: always()
        uses: actions/upload-artifact@v7
        with:
          name: drift-report
          path: drift.json
```

Every morning at six (UTC), GitHub starts a fresh machine, installs sadame, loads an SSH key from the repository's secrets, and runs the check. A drift fails the job, which sends the usual failure email. The summary page shows the Markdown report, and the JSON is kept as an artifact for thirty days.

What the job needs:

- `SSH_PRIVATE_KEY`: a private key for a key pair made for CI. Its public half goes into `authorized_keys` of the users in `hosts.rb`.
- `SSH_KNOWN_HOSTS`: the output of `ssh-keyscan web1 web2 db1` (with your servers' names), so the job checks the servers' identities instead of trusting whatever answers.
- Passwordless sudo for those users, since drift reads files only root can read.



The servers must be reachable from GitHub's machines. For servers in a private network, run the check from a machine inside it, with `cron` or a systemd timer, and send the report where your team will see it.

For a new project, `sadame init --ci github` writes the workflow along with the starting recipe; `--ci gitlab` writes the same for GitLab CI:

```
$ sadame init infra --ci github
created infra/recipe.rb
created infra/vars.rb
created infra/hosts.rb
created infra/templates/motd.erb
created .github/workflows/sadame-drift.yml

Next:
  sadame validate infra        check the recipe without connecting to anything
  sudo sadame plan infra       see what it would change on this machine
  sadame plan infra -g all     on your servers, once infra/hosts.rb lists them
```

What you learned

- `sadame drift` exits `0`, `1` or `2`, which makes it a check scripts and CI can run.
- `--json` and `sadame report` turn a drift run into Markdown or JUnit XML.
- Repair drift with `apply`, or accept it by changing the recipe.
- A scheduled CI job runs the check every day and fails loudly.

Try it

1. Run `sadame drift site.rb -g all --json | head -n 30` and find the `status` of each host.
2. Stop nginx on web2, run the drift check with `--tags web` and then with `--tags app`. Which one finds it?
3. Write the drift check as a systemd timer on the practice machine, with a `file` for the `.service` unit, a `file` for the `.timer` unit, and a `service` that enables the timer.

Chapter 19. Safety Nets

Infrastructure code changes real machines, so its mistakes cost more than a failed test. This chapter collects the habits and tools that catch mistakes early, from the moment you type a recipe to the moment it reaches the last server, and adds the two you have not used yet: `sadame validate` in depth, and the language server that brings its checks into your editor.

19.1. Layers of safety

Every layer catches a different kind of mistake, at a different cost:

Layer	Catches	When
The editor (`sadame lsp`)	typos in options, unknown resources, Ruby errors, missing notification targets	while you type
`sadame validate`	the same, for every host, with its vars and platform	before any server is involved
`sadame plan`	what the machine says: a package that does not exist, a file that differs	before changing anything
`verify:`	a configuration file the program itself rejects	before replacing the file
`--serial`	a change that breaks a server	after the first batch
`sadame drift`	changes made outside the recipe	every day
git	everything else: who changed what, and the way back	always

The first two need no server and take a fraction of a second. Use them all the time.

19.2. validate

`validate` evaluates a recipe the way `plan` does, but against a stand-in machine that runs no commands. For each host, it builds the vars from the inventory, runs the recipe's Ruby, checks every resource's options, templates and notifications, and counts the resources. It connects to nothing.

Make a typo in a copy of the fleet's recipe, `mdoe:` for `mode:` in the web role, and validate:

```
$ sadame validate site.rb -g all
sadame 0.5.1 validate site.rb, as on Debian GNU/Linux 12 (bookworm)

  web1  error  roles/web.rb:18: file[/etc/nginx/sites-available/lab-site]: unknown option :mdoe (valid options:
content, group, locals, mode, notifies, owner, sensitive, source, state, subscribes, tags, template, verify)
  web2  error  roles/web.rb:18: file[/etc/nginx/sites-available/lab-site]: unknown option :mdoe (valid options:
content, group, locals, mode, notifies, owner, sensitive, source, state, subscribes, tags, template, verify)
  db1   ok     15 resources

validate on 3 hosts: 1 ok, 2 error.

$ echo $?
2
```

The mistake is found for each host that includes the web role, with the file, the line of the misspelled option, and the list of valid options. db1 does not include that role and is fine. The exit status is `2`, which makes `validate` a good first step for CI on every pull request: it needs no SSH key and no servers.

By default the stand-in machine is Debian. For hosts that run something else, say so with `--platform`:

```
$ sadame validate site.rb -g db --platform rocky
sadame 0.5.1 validate site.rb, as on Rocky Linux 9.4 (Blue Onyx)

  db1  ok     15 resources

validate on 1 host: 1 ok.
```

With `--platform rocky`, `platform.family` is `"rhel"`, so the recipe takes its Rocky branches: `cronie`, `sshd`, `dnf-automatic`.

What `validate` cannot know is what only a real machine knows: whether a package exists in its repositories, whether a service is installed, what a file contains. That is what `plan` is for.

19.3. verify: the last check

`plan` cannot tell whether a configuration file is valid for the program that reads it. `verify:` can, at the last possible moment: after sadame has written the new content next to the file, before it replaces the file. Add a setting sshd does not know, `"Bogus" ⇒ "yes"`, to the Hash in a copy of the fleet's hardening role, and apply it to web1:

```
$ sadame apply site.rb -g all --limit web1 --tags hardening
[web1] sadame 0.5.1 apply on web1 (Debian GNU/Linux 12 (bookworm))
[web1] Only resources tagged hardening: 4 of 22.
[web1]
[web1] failed     file[/etc/ssh/sshd_config.d/00-hardening.conf]
[web1]            --- /etc/ssh/sshd_config.d/00-hardening.conf
[web1]            +++ /etc/ssh/sshd_config.d/00-hardening.conf (desired)
[web1]            @@ -1,2 +1,3 @@
[web1]             PermitRootLogin no
[web1]             MaxAuthTries 3
[web1]            +Bogus yes
[web1]            error: file[/etc/ssh/sshd_config.d/00-hardening.conf]: verify failed,
/etc/ssh/sshd_config.d/00-hardening.conf was not changed: sshd -t -f /etc/ssh/sshd_config.d/00-
hardening.conf.sadame-new
[web1]            /etc/ssh/sshd_config.d/00-hardening.conf.sadame-new: line 3: Bad configuration option: Bogus
[web1]            /etc/ssh/sshd_config.d/00-hardening.conf.sadame-new: terminating, 1 bad configuration options
[web1] skipped    service[ssh]
[web1] skipped    package[unattended-upgrades]
[web1] skipped    file[/etc/apt/apt.conf.d/20auto-upgrades]
[web1]
[web1] Summary
[web1]   changed      0
[web1]   unchanged    0
[web1]   failed       1  file[/etc/ssh/sshd_config.d/00-hardening.conf]
[web1]   skipped      3  service[ssh]
[web1]                   package[unattended-upgrades]
[web1]                   file[/etc/apt/apt.conf.d/20auto-upgrades]
[web1]   notified     0
[web1]
[web1] Finished in 0.01s.

Hosts
  web1  failed
```

```
apply on 1 host: 1 failed. Finished in 0.22s.

$ ssh web1 cat /etc/ssh/sshd_config.d/00-hardening.conf
PermitRootLogin no
MaxAuthTries 3
```

`sshd -t` rejected the new file, so the old one stayed, and the error includes sshd's own explanation, `Bad configuration option: Bogus`. The resources after it were skipped, as after any failure. sshd never saw the broken configuration and kept running with the old one, so the next SSH connection to the server still works. Use `verify:` on every complete configuration file whose program has a check mode: `visudo -cf %{path}`, `sshd -t -f %{path}`, `haproxy -c -f %{path}`.

19.4. The editor

`sadame lsp` is a **language server**: a program your editor starts in the background and talks to while you work. It is in the same `sadame` binary. While you edit a recipe, it:

- underlines what `validate` would report, on the line it is on, as you type;
- completes resource names, their options, `state:` values, and fact names after `facts[:`;
- shows the documentation of a resource, option or fact when you point at it;
- lists the recipe's resources in the editor's outline.

It checks a file through the recipe that includes it, for every host in the nearest `hosts.rb`: a web role is checked with the web servers' vars, not the database server's.

Editors that speak the Language Server Protocol need a few lines of configuration. For Neovim 0.11 and later:

code/19-safety-nets/editors/neovim.lua

```
-- sadame's language server in Neovim 0.11 and later: put this in
-- ~/.config/nvim/init.lua, or in a file it loads.
vim.lsp.config("sadame", {
  cmd = { "sadame", "lsp" },
  filetypes = { "ruby" },
  root_markers = { "hosts.rb", "vars.rb", "recipe.rb", "site.rb" },
})
vim.lsp.enable("sadame")
```

For Helix:

code/19-safety-nets/editors/helix-languages.toml

```
# sadame's language server in Helix: add this to ~/.config/helix/languages.toml.
# List the Ruby servers you already use after "sadame".
[language-server.sadame]
command = "sadame"
args = ["lsp"]

[[language]]
name = "ruby"
language-servers = ["sadame", "ruby-lsp"]
```

For Emacs with Eglot:

code/19-safety-nets/editors/emacs.el

```
;; sadame's language server in Emacs, with Eglot.
(with-eval-after-load 'eglot
  (add-to-list 'eglot-server-programs '((ruby-mode ruby-ts-mode) "sadame" "lsp")))
```

With the typo from above, the editor underlines the `mdoe:` line with the message you saw from `validate`: `unknown option :mdoe (valid options: …)`. The server activates only in directories that have a `recipe.rb`, `site.rb`, `vars.rb` or `hosts.rb`, so other Ruby projects are left alone, and it works next to a Ruby language server such as Ruby LSP.

19.5. Habits

Tools catch mistakes; habits keep them small.

Plan before apply, every time. Read the plan as you would read a diff in a code review. A plan with more changes than you expected is the most valuable warning you will get.

Small changes. One role, one setting, one release per change. When something breaks, you know which change broke it.

Roll out gradually. `--limit` the first server, or `--serial 1` the group. Canary hosts, as `tier: "canary"` in chapter 11, make it a routine.

Recipes in git. Every change reviewed, every change revertable. The plan of a change belongs in its pull request.

Never fix by hand without telling the recipe. If you must edit a server during an incident, put the change into the recipe the same day. The drift check will remind you if you forget.

Keep secrets out of the repository. Read them from the environment, as in chapter 15, and mark secret files `sensitive: true`.

What you learned

- Safety comes in layers, from the editor to the daily drift check.
- `validate` checks every host's recipe without a server; `--platform` checks it as on another system.
- `sadame lsp` brings the same checks, completion and documentation into Neovim, Helix, Emacs and other editors.
- Plan, small changes, gradual rollouts and git keep the mistakes that slip through small.

Try it

1. Set up `sadame lsp` in your editor, open a recipe from `~/code`, and misspell an option.
2. Run `sadame validate site.rb -g all --platform alpine` on the fleet. Every host is `ok`. Why does that not mean the fleet would work on Alpine? (Look at the `else` branch of the hardening role.)

3. Add `sadame validate` to a CI workflow that runs on every pull request.

Chapter 20. When Things Go Wrong

Sooner or later a run fails. Sadame's error messages are written to answer three questions: **where** (which host, which file and line, which resource), **what** (the exact error, often the command's own output), and **what to do**. This chapter shows the failures you will meet most often, each made on purpose in the lab, and how to read them.

```
$ cd ~/code/20-troubleshooting
```

20.1. Three kinds of failure

Failures come in three kinds, and the kind tells you where to look:

Recipe errors

The recipe itself is wrong: a misspelled option, a missing template, a notification to nothing. Sadame stops before touching anything, with exit status `2`. Fix the recipe.

Resource failures

The recipe is fine, but the machine refused: a package that does not exist, a command that failed. Sadame reports the resource as failed, stops that host's run, and exits `1`. Fix the recipe or the machine.

Connection failures

Sadame could not reach a host, log in, or use sudo there. The host is `unreachable` or `error` in the summary, the others run, and the exit status is `2`. Fix SSH, the inventory, or sudo.

20.2. A misspelled option

code/20-troubleshooting/unknown-option.rb

```
# A typo in an option name: "contents" instead of "content".
file "/tmp/report.txt", contents: "weekly report\n"
```

```
$ sadame plan unknown-option.rb
sadame: unknown-option.rb:2: file[/tmp/report.txt]: unknown option :contents (valid options: content, group,
locals, mode, notifies, owner, sensitive, source, state, subscribes, tags, template, verify)

$ echo $?
2
```

File, line, resource, the wrong option, and every option that would have been right. In most other tools, an unknown option is silently ignored, and you find out when the file is not as you expected. Sadame refuses to run a recipe it does not fully understand. `validate` and the editor report the same.

20.3. A notification to nowhere

code/20-troubleshooting/missing-notification.rb

```
# A notification for a resource that is not declared: the service is
# called nginx, not ngnix.
file "/etc/nginx/conf.d/extra.conf",
     content: "# extra settings\n",
     notifies: [:reload, "service[ngnix]"]

service "nginx", state: :running
```

```
$ sadame plan missing-notification.rb
sadame: file[/etc/nginx/conf.d/extra.conf] notifies service[ngnix], which is not declared
```

`ngnix` instead of `nginx`. Without this check, the configuration would change and nginx would quietly never reload. Sadame finds it when it loads the recipe, before any resource runs.

20.4. A template error

code/20-troubleshooting/template-error.rb

```
# A template that uses a var nobody set: there is no vars.rb here, so
# vars[:workers] is nil, and nil cannot be multiplied.
file "/tmp/app.conf", template: "app.conf.erb"
```

code/20-troubleshooting/templates/app.conf.erb

```
port = <%= vars[:port] %>
workers = <%= vars[:workers] * 2 %>
```

```
$ sadame plan template-error.rb
sadame 0.5.1 plan on local (Debian GNU/Linux 12 (bookworm))

x file[/tmp/app.conf]
    error: templates/app.conf.erb:2: undefined method '*' for NilClass

Plan: 0 to create, 0 to update, 0 to remove, 0 to run, 0 unchanged, 1 failed.
```

The error points into the template, `app.conf.erb:2`, not the recipe. `undefined method '' for NilClass` is Ruby's way of saying "this value is nil": `vars[:workers]` was never set, because there is no `vars.rb` here. Note that line 1 did *not* fail: `vars[:port]` is `nil` too, and `<%= nil %>` prints nothing. A missing var shows up either as an error or as an empty value, depending on what the template does with it.



For a var that must be set, use `vars.fetch(:workers)`. A missing key then fails with `Key not found: :workers`, which names the culprit.

20.5. A package that does not exist

code/20-troubleshooting/no-such-package.rb

```
# A package name that does not exist in the distribution.
package "nginx-full-extra"
```

```
$ sudo sadame apply no-such-package.rb
sadame 0.5.1 apply on local (Debian GNU/Linux 12 (bookworm))

failed     package[nginx-full-extra]
           state: absent -> installed
           error: command failed (exit 100): DEBIAN_FRONTEND=noninteractive apt-get install -y nginx-full-extra
           E: Unable to locate package nginx-full-extra

Summary
  changed      0
  unchanged    0
  failed       1  package[nginx-full-extra]
  skipped      0
  notified     0

Finished in 0.75s.

$ echo $?
1
```

A resource failure. Sadame prints the command it ran, its exit status, and the package manager's own error, `Unable to locate package`. Package names differ between distributions and versions; `apt-cache search` and `dnf search` find the right one. `plan` had no way to know: asking apt whether a package is installed does not say whether it exists.

When a resource fails, the run on that host stops. The resources after it are reported as `skipped`, because they might depend on it: there is no point writing nginx's configuration if nginx failed to install. The summary lists what failed and what was skipped, and the next `apply` picks up where this one stopped.



Delayed notifications do not run after a failure either. If a configuration file changed before the failure, its reload is lost: on the next run, the file is unchanged and notifies nothing. After fixing a failed run, check whether a service still runs with its old configuration, and reload or restart it yourself.

20.6. Hosts that cannot be reached

The last example has an inventory with four kinds of host: one that works, one whose user must type a password for sudo, one whose name does not exist, and one where the user has no key:

code/20-troubleshooting/hosts.rb

```
# Servers that cannot all be reached, for chapter 20.
host "web1", user: "deploy", sudo: true
host "web2", user: "intern", sudo: true
host "web3", user: "deploy", sudo: true
host "db1", user: "admin", sudo: true
```

(web2 has a user `intern`, made for this chapter, who is in the `sudo` group, so sudo asks for a password.)

```
$ sadame plan motd.rb --limit '*'
```

```
[web1] sadame 0.5.1 plan on web1 (Debian GNU/Linux 12 (bookworm))
[web1]
[web1] ~ file[/etc/motd]
[web1]     --- /etc/motd
[web1]     +++ /etc/motd (desired)
[web1]     @@ -1,2 +1 @@
[web1]     -web1 (web, all), managed by sadame.
[web1]     -Changes made by hand will be undone. Questions: ops@example.com
[web1]     +Managed by sadame.
[web1]
[web1] Plan: 0 to create, 1 to update, 0 to remove, 0 to run, 0 unchanged.

Hosts
  web1  ok
  web2  error        sudo failed: sudo: a password is required
  web3  unreachable  connect failed: ssh: Could not resolve hostname web3: Name or service not known
  db1   unreachable  connect failed: admin@db1: Permission denied (publickey,gssapi-keyex,gssapi-with-mic).

plan on 4 hosts: 1 ok, 1 error, 2 unreachable. Finished in 0.22s.

$ echo $?
2
```

web1 planned as usual; the other three failed in different ways, each with the underlying message:

Message	What to do
`sudo failed: sudo: a password is required`	Give the user passwordless sudo for deployments, with a line like `deploy ALL=(ALL) NOPASSWD: ALL` in `/etc/sudoers.d/deploy`, or run without sudo.
`Could not resolve hostname`	A typo, or a name DNS does not know. Set `address:` in `hosts.rb`, or `HostName` in `~/.ssh/config`.
`Permission denied (publickey,…)`	Wrong user, or the server does not have your key. Check the user, and copy the key with `ssh-copy-id USER@HOST`.
`Host key verification failed`	The server's key changed. If you rebuilt the server, `ssh-keygen -R HOST` and connect once; if you did not, find out why before you do anything else.
`Connection refused`	sshd is not running, or listens on another port: set `port:` or `Port`.

The first thing to try is always the plain `ssh` command sadame would run: `ssh admin@db1 true`. Sadame uses your `ssh`, so if that fails, sadame fails the same way, and `ssh -v` tells you more than any tool built on top of it can.

20.7. A checklist

When a run fails and the message is not enough:

1. **Read the summary first.** Which hosts, which kind of failure?
2. **Plan the one failing host**, with `--limit`, for a shorter output.
3. **Validate**, to rule out the recipe: `sadame validate site.rb -g all`.
4. **Run the command by hand** on the server: the exact command is in the error.

5. **Read the facts** the recipe sees there: `sadame facts -h HOST`.
6. **Look at the vars**: `sadame hosts --limit HOST` shows each one and its source.

What you learned

- Recipe errors stop everything with status `2`; a resource failure stops that host's run and skips the rest, with status `1`; connection failures mark the host and leave the others running.
- Error messages carry the file and line, the resource, and the underlying command's output.
- `nil` errors in templates mean a var was not set; `vars.fetch` makes that explicit.
- For connection problems, run the plain `ssh` command first.

Try it

1. Change `template-error.rb` to use `vars.fetch(:workers)` in the template, and plan. Compare the message.
2. Run `ssh intern@web2 sudo -n true` and read sudo's own message.
3. Make a recipe with two packages, the first one misspelled, and apply it. What does the summary say about the second one?

Chapter 21. Where to Go Next

You started this book with a file in `/tmp` and finished it with a fleet: a website, a database, an application with rolling upgrades, hardening, and a daily drift check, all from one repository of plain Ruby. This last chapter is about taking that to your own servers, and about what to learn next.

21.1. From the lab to your servers

Everything in this book works the same on real servers. What changes is around the recipe:

The servers

Any Linux server with SSH, a shell and `tar`: a cloud VPS, a virtual machine at work, a Raspberry Pi. Debian, Ubuntu, Rocky, AlmaLinux, Fedora, Alpine and Arch are supported, on x86_64 and ARM64.

The account

A user with your SSH key and passwordless sudo, as `deploy` in the lab. Many cloud images give you `root` only; the first recipe you run on such a server should create that user, and the second should forbid root logins.

The secrets

In your password manager or your cloud's secret store, read into the environment of the machine running sadame (chapter 15).

The repository

Your recipes, under git, reviewed like code.

Sadame's documentation has a guide, *Provision a new VPS*, that takes a fresh Debian or Ubuntu server, with only `root` and your key, to a hardened web server in three runs: an administrator account; SSH on a different port with keys only, a firewall, fail2ban and automatic updates; then nginx. Every step is an example from sadame's repository, used as it is. It is the natural next exercise after this book.

21.2. Working as a team

When more than one person changes the same servers, the habits of chapter 19 become a workflow, which the guide *Team workflow* sets up with CI:

1. Every change is a pull request.
2. CI runs `sadame validate` and `sadame plan` on the pull request, and puts the plan on the run's summary page. Reviewers read the plan along with the code.
3. After merging, a maintainer applies, or CI does, after an approval.
4. The daily drift check runs on the main branch.

The result is that the repository is always the truth about the servers, and every change to a server has an author, a review and a date.

21.3. Sharing recipes

A role that several projects need, such as your team's SSH hardening, can live in its own git repository and be added to each project as a git submodule:

```
$ git submodule add https://github.com/example/sadame-hardening.git site/vendor/hardening
$ git -C site/vendor/hardening checkout v1.2.0
```

Then `include_recipe "vendor/hardening/hardening"` uses it like any other role. Each project pins the version it uses and upgrades on purpose, with the update reviewed as a change. Sadame needs nothing special for this: an included recipe is a file in the project, wherever it came from.

21.4. Coming from another tool

If you already have Ansible playbooks or Itamae recipes, sadame's documentation has migration guides for both. Most of an Ansible playbook maps resource by resource:

Ansible	sadame
playbook (YAML)	recipe (Ruby)
`group_vars`, `host_vars`, `-e`	group and host vars in `hosts.rb`, `--var`
Jinja2 templates	ERB templates
`notify:` and handlers	`notifies:` and `subscribes:`
`when:`	Ruby `if`
`loop:`	Ruby `each`
`--check --diff`	`sadame plan`
`serial:`	`--serial`

Itamae's recipes are closer still: same language, many of the same resource names. The biggest differences are sadame's inventory, its `plan` and `drift`, and that it copies itself to the server instead of needing Ruby there.

21.5. Your own computer

Recipes are not only for servers. The setup of your own Linux computer, the packages you add, your dotfiles, a timer or two, is usually done by hand and remembered by nobody. A recipe keeps it: a fresh installation plus `sadame apply` gives you the same machine, and `sadame drift` answers "what did I change?" in one command. Sadame's documentation has a guide for Omarchy, an Arch Linux desktop.

21.6. When sadame is not the right tool

Sadame is young and deliberately small. Choose something else when:

- your machines run Windows, or you need to manage network equipment;
- you need thousands of ready-made modules for cloud services and appliances: Ansible's

ecosystem is vast;

- you need a central server that every machine checks in with on a schedule, with reporting across thousands of nodes: Puppet and Chef were built for that;
- you want to create the servers themselves, networks and load balancers: that is a job for an infrastructure-as-code tool such as OpenTofu or Terraform, which works well together with sadame. One creates the machines, the other configures them.

21.7. Keep learning

* **Sadame's documentation**, https://enderkus.github.io/sadame/: every resource with all its options, the command line, JSON output, and the guides mentioned in this chapter. * **The source**, https://github.com/enderkus/sadame: the examples directory has a runnable recipe for most situations, tested in CI. * **This book's repository**, https://github.com/enderkus/sadame-beginner-guide: the lab and every recipe of the book. Issues and corrections are welcome. * **News**: sadame is on X as `@sadame_rb`.

If sadame helped you, star it on GitHub: it is how other people find it.

21.8. A last word

The word **sadame** (定め) means a rule, a decision, something settled. A recipe is exactly that: a decision about how a machine should be, written down where everyone can read it, checked every day, and changed on purpose. Servers come and go. Decisions written down stay.

Thank you for reading.

Appendix A: Command Reference

A summary of sadame 0.5.1's command line, for looking things up. The full reference is in sadame's documentation.

A.1. Synopsis

```
sadame COMMAND [RECIPE] [options]
```

The recipe is a file such as `site.rb`, or a directory, which means its `recipe.rb`; without one, `./recipe.rb`. `vars.rb` and `hosts.rb` next to the recipe are loaded when they exist. Options may come before or after the command.

A.2. Commands

Command	Changes machines	What it does
`plan`	no	Reads the state of every resource and prints what `apply` would change, as diffs.
`apply`	yes	Changes what differs from the recipe, in order, then runs notifications.
`drift`	no	The same comparison as `plan`; exits `1` when anything differs.
`validate`	no	Evaluates the recipe for each selected host against a stand-in machine and renders its templates. Connects to nothing.
`hosts`	no	Prints the selected hosts, their connection options, and their vars with where each came from.
`facts`	no	Prints the facts recipes see, here or on the selected hosts.
`init [DIR]`	no	Writes a starting project: `recipe.rb`, `vars.rb`, `hosts.rb`, `templates/motd.erb`.
`report FILE`	no	Turns the `--json` output of `plan`, `apply` or `drift` into Markdown or JUnit XML.
`lsp`	no	Runs the language server for editors.

A.3. Choosing hosts

Option	Value	Effect
`-h`, `--host`	`[USER@]HOST[:PORT]`, comma-separated	These hosts. A host from the inventory keeps its settings.
`-g`, `--group`	group names, comma-separated	The hosts of these groups.

Option	Value	Effect
`--limit`	a pattern such as `"web*"`	Only the selected hosts whose names match; alone, filters the whole inventory.
`--inventory`	path	Another inventory instead of `hosts.rb`.
`--sudo`		Run sadame with passwordless sudo on every target.
`--forks`	number	At most this many hosts at once.
`--serial`	number or percentage	Batches, one after another; stops after a batch with a failure.

Without any of them, sadame runs on the machine it was started on.

A.4. Other options

Option	Effect
`--var KEY=VALUE`	Sets `vars[:KEY]`, over every other source. Repeatable. Whole numbers become Integers, `true` and `false` booleans.
`--vars-file PATH`	More vars from a Ruby file that returns a Hash. Local runs only.
`--tags T,…`	Only resources with one of these tags.
`--skip-tags T,…`	Every resource except those with one of these tags.
`--platform NAME`	`validate`: as on `debian` (the default), `ubuntu`, `rocky`, `alpine` or `arch`.
`--format FORMAT`	`report`: `markdown` (the default) or `junit`.
`--ci SYSTEM`	`init`: also write a daily drift check for `github` or `gitlab`.
`--json`	One JSON document instead of text.
`--color`, `--no-color`	Force colors on or off.
`--host-name NAME`, `--host-groups G,…`	`host_name` and `host_groups` for local runs.
`--version`, `--help`	The version, or a short help.

A.5. Exit status

Status	Meaning
`0`	Done: the plan was computed, every change succeeded, nothing drifted.
`1`	`apply`: a resource or notification failed. `drift`: something differs.
`2`	Sadame could not do its job: a wrong command line, a recipe error, a state it could not read, a host it could not reach.

With several hosts, the run exits with the highest status of any host.

A.6. In a recipe

Method	Meaning
`vars`	The merged vars: `vars.rb`, groups, host, `--var`.
`facts`	What the machine has: `facts[:cpus]`, `facts[:memory_mb]`, … (appendix C).
`platform`	The operating system: `family`, `package_manager`, `id`, `version`, `codename`, `name`.
`host_name`, `host_groups`	The machine's name in the inventory, and its groups.
`include_recipe "path", tags: […]`	Adds another recipe file's resources here.

A.7. In hosts.rb

Method	Meaning
`group :name do … end`	A group; add `include: [:a, :b]` to take in other groups.
`host "name", …`	A host, with `address:`, `user:`, `port:`, `sudo:`, `ssh_opts:` and `vars:`.
`vars key: value`	Vars for every host of the group.
`command_output("cmd")`	Runs a command on the machine running sadame and returns its output.

Appendix B: Resource Reference

Sadame 0.5.1 has ten resource types. Every one of them also takes `notifies:`, `subscribes:` and `tags:`; they are not repeated in the tables. "Unmanaged" means sadame leaves that property as it finds it unless you give the option.

B.1. file

`file "PATH", …`: a file's content, permissions and owner, or its absence.

Option	Meaning
`content:`	The exact content.
`template:`	An ERB template, with `vars`, `locals` and `facts`. Found next to the recipe, then in `templates/`.
`source:`	A file copied as it is. Found next to the recipe, then in `files/`.
`locals:`	A Hash for the template: `locals[:name]`.
`mode:`	Permission bits, such as `"0644"`. Unmanaged by default.
`owner:`, `group:`	Owner and group names. Unmanaged by default.
`state:`	`:present` (default) or `:absent`.
`verify:`	A command that checks the new content first; `%{path}` is the new file.
`sensitive:`	`true` hides the content's diff in the output.

B.2. directory

Option	Meaning
`mode:`, `owner:`, `group:`	As for `file`.
`state:`	`:present` (default) or `:absent`.
`recursive:`	With `:absent`, remove the directory and everything in it.

B.3. link

Option	Meaning
`to:`	Where the link points. Required. Relative targets are relative to the link's directory.
`state:`	`:present` (default) or `:absent`.
`force:`	Replace a regular file that is where the link should be.

B.4. line

`line "TEXT", path: "FILE", …`: one line in a file someone else owns.

Option	Meaning
`path:`	The file. Required.
`line:`	The line, when the name is only a description.
`match:`	A Regexp: matching lines are replaced by the line; if none matches and the line is missing, it is appended.
`state:`	`:present` (default) or `:absent`, which also removes lines that match `match:`.
`verify:`, `sensitive:`	As for `file`.

B.5. package

Option	Meaning
`state:`	`:installed` (default) or `:absent`.
`options:`	Extra flags for the install or remove command.

The package manager follows the platform: apt, dnf, apk or pacman.

B.6. service

Option	Meaning
`enable:`	`true` or `false`: start at boot. Unmanaged by default.
`state:`	`:running` or `:stopped`. Unmanaged by default.
`user:`	Manage a unit in this user's own systemd instance.

Actions for notifications: `:start`, `:stop`, `:restart`, `:reload`, `:enable`, `:disable`. Works with systemd and OpenRC.

B.7. user

Option	Meaning
`uid:`, `home:`, `shell:`	Unmanaged by default.
`groups:`	Groups the user must be in; other memberships are left alone. The groups must exist.
`system:`	`true` creates a system account, without a home directory.

Option	Meaning
`state:`	`:present` (default) or `:absent`, which keeps the home directory.

B.8. group

Option	Meaning
`gid:`	The numeric group ID.
`system:`	`true` creates a system group.
`state:`	`:present` (default) or `:absent`.

B.9. exec

`exec "NAME", ⋯`: a shell command, with guards.

Option	Meaning
`command:`	The command, when the name is a label. Default: the name.
`creates:`	Skip when this path exists.
`not_if:`	Skip when this command succeeds.
`only_if:`	Skip unless this command succeeds.
`cwd:`	Working directory for the command and its guards.
`env:`	Environment variables for the command and its guards.
`action:`	`:run` (default) or `:nothing`: only when notified.

Action for notifications: `:run`.

B.10. git

`git "DIRECTORY", ⋯`: a checkout kept at a revision.

Option	Meaning
`repository:`	The URL to clone. Required.
`revision:`	A tag, a branch or a full commit. Default: the remote's default branch.
`user:`	Run git as this user, who then owns the checkout.
`state:`	`:present` (default) or `:absent`.

B.11. Notifications

```
notifies: [:reload, "service[nginx]"]                      # delayed: once, at the end
notifies: [:run, "exec[systemctl daemon-reload]", :immediately]
notifies: [[:reload, "service[nginx]"], [:run, "exec[...]"]]
subscribes: [:restart, "git[/srv/app]"]                     # the other direction
```

Only a resource that **changed** during `apply` notifies. A failed resource stops the run, and delayed notifications are then skipped.

Appendix C: Facts and Platforms

C.1. Facts

`facts[:name]` in a recipe or template. Each fact is read the first time it is asked for, on the machine the recipe runs on. A fact that cannot be read is `nil` (an empty Array for the address lists); a misspelled name is an error.

Fact	Type	Example
`hostname`	String	`"web1"`
`fqdn`	String	`"web1.example.com"`
`arch`	String	`"x86_64"`, `"aarch64"`
`kernel`	String	`"6.1.0-26-amd64"`
`cpus`	Integer	`4`
`memory_mb`	Integer	`7941`
`swap_mb`	Integer	`2047`
`root_disk_gb`	Integer	`39`
`ipv4`	String or nil	`"203.0.113.10"`: the address of the route to the internet
`default_interface`	String or nil	`"eth0"`
`ipv4_addresses`	Array	every IPv4 address except loopback
`ipv6_addresses`	Array	global IPv6 addresses
`virtualization`	String or nil	`"kvm"`, `"docker"`; nil on bare metal
`container`	`true` or `false`	`false`
`service_manager`	String or nil	`"systemd"`, `"openrc"`
`machine_id`	String or nil	`/etc/machine-id`

`sadame facts` prints them for this machine; with `-h`, `-g` or `--limit`, for each selected host; with `--json`, as JSON.

C.2. Platform

Method	Values
`platform.family`	`"debian"`, `"rhel"`, `"alpine"`, `"arch"`, or `nil`
`platform.package_manager`	`"apt"`, `"dnf"`, `"apk"`, `"pacman"`, or `nil`
`platform.id`	`ID` from `/etc/os-release`: `"debian"`, `"ubuntu"`, `"rocky"`, …

Method	Values
`platform.version`	`VERSION_ID`: `"12"`, `"24.04"`, `"9.4"`
`platform.codename`	`VERSION_CODENAME`: `"bookworm"`
`platform.name`	`PRETTY_NAME`: `"Debian GNU/Linux 12 (bookworm)"`

C.3. Names that differ between families

Tested in this book's lab, on Debian 12 and Rocky Linux 9:

What	Debian, Ubuntu	Rocky, Alma, Fedora
cron package / service	`cron` / `cron`	`cronie` / `crond`
SSH service	`ssh`	`sshd`
SQLite command-line tool	`sqlite3`	`sqlite`
PostgreSQL server	`postgresql`	`postgresql-server`, then `postgresql-setup --initdb`
PostgreSQL client	`postgresql-client`	`postgresql`
Automatic updates	`unattended-upgrades`	`dnf-automatic`, with `dnf-automatic-install.timer`
Web server's site directory	`/etc/nginx/sites-available/`	`/etc/nginx/conf.d/`

Appendix D: Glossary

apply

The command that changes a machine to match its recipe, in the recipe's order, then runs notifications.

configuration management

Describing how machines should be set up, in files, and letting a tool make them so and keep them so.

desired state

What a recipe declares: not the steps, but the result.

drift

A difference between a machine and its recipe that appeared after the last apply, usually from a change made by hand. Also the command that reports it.

facts

What a machine has, read on the machine: CPUs, memory, addresses.

group

A named set of hosts in the inventory.

guard

An `exec` option, `creates:`, `not_if:` or `only_if:`, that decides whether its command needs to run.

host

A machine in the inventory, with how to reach it.

idempotence

The property that applying a recipe a second time changes nothing.

inventory

The list of hosts and groups, `hosts.rb`.

locals

Values a recipe passes to one template with `locals:`.

mruby

A small implementation of Ruby, made to be embedded. Sadame is built with it, which is why it is one file with no dependencies.

notification

A link between resources: when one changes, run an action on the other.

plan

The command that shows what `apply` would change, without changing anything.

platform

The operating system a recipe runs on: its family, name and version.

push mode

Running a recipe on remote machines over SSH, by copying sadame and the recipe there for the run.

recipe

A Ruby file that declares resources.

resource

One thing on a machine that a recipe manages: a file, a package, a service. Written `type[name]`.

role

A recipe file for one job a server can have, included from `site.rb`.

rolling upgrade

Changing a group of servers in batches, so that some keep working while others change.

tag

A label on resources, to run part of a recipe.

template

An ERB file that becomes a file's content.

vars

The values a recipe reads, merged from `vars.rb`, groups, hosts and `--var`.

Index

@

--limit, 48
--serial, 75
  batches, 76
--skip-tags, 57
--sudo, 44
--tags, 57
--var, 27
-g option, 48
-h option, 43
  several hosts, 45

A

agent, 5
alternatives, 99
Ansible, 99
apply, 5, 11, 110
  after a failure, 95
  order, 35
architecture, 46

C

certificates, 64
CI, 86
command_output, 51
commands, 101
configuration drift, 5
configuration management, 110

D

daemon-reload, 73
dependencies, 40
desired state, 4
directory resource, 30, 104
  recursive, 30
drift, 5, 12, 83, 110
  exit status, 84
  permissions, 22
  services, 35
drop-in files, 79

E

editors, 90
ENV, 68
ERB, 25
errors
  connection, 95
  kinds, 93
  notification, 94
  resource failure, 94
  templates, 94
  unknown option, 93
exec resource, 39, 106
  action nothing, 40
  creates, 67
  guards, 39
exit status, 12, 13, 84, 102

F

facts, 52, 108, 110
  drift, 55
  in templates, 67
  reading, 53
file resource, 10, 20, 104
  atomic write, 22
  mode, 21
  owner, 21
  template, 25
  verify, 22

G

git resource, 73, 106
git submodules, 99
GitHub Actions, 86
group resource, 39, 106
guards, 39, 110

H

habits, 91
hardening, 79
hello-app, 72
host_groups, 48
host_name, 43
hosts.rb, 47

I

idempotence, 5, 11, 110
include_recipe, 56
  once, 57
installing sadame, 9
inventory, 47, 110
  dynamic, 51
  groups, 47, 110
Itamae, 99

J

JSON output, 13, 84
JUnit, 85

L

lab, 7
  entering, 8
  machines, 7
  resetting, 8
  starting, 8
language server, 90
line resource, 31, 105
  match, 31, 67
  verify, 31, 89
link resource, 30, 104
  force, 30
locals, 26

M

mruby, 110

N

nginx, 61
  reload timing, 63
  sites-enabled, 61
nil, 94
notifications, 33, 106, 110
  actions, 36
  deduplicated, 63
  delayed, 35
  exec, 40
  immediately, 36, 73
notifies, 33

P

package manager, 19
package resource, 19, 105
  options, 20
parallel runs, 45
passwords, 68
plan, 5, 10, 110
  limits, 69
platform, 53, 108, 111
  package names, 109
  unsupported, 55
platform.family, 53
production, 98
provisioning, 98
pull requests, 98
push mode, 5, 43, 111
  how it works, 46

R

recipe, 4, 10, 111
regular expressions, 31
resource, 4, 10
resources, 100, 111
  directory, 30
  exec, 39
  file, 20
  git, 73
  group, 39
  line, 31
  link, 30
  package, 19
  service, 33
  user, 39
roles, 56, 111
rolling upgrades, 75, 111
Ruby, 15
  arrays, 17
  conditions, 17
  hashes, 17
  loops, 17
  methods, 17
  strings, 16
  symbols, 16
  variables, 16

S

sadame apply, 11
sadame drift, 12
sadame facts, 52
sadame hosts, 47
sadame init, 59
  --ci, 87
sadame lsp, 90
sadame plan, 10
sadame report, 84
sadame validate, 59, 80, 88
secrets, 68
sensitive, 68
service resource, 33, 105
sharing recipes, 99
side effects, 18
site.rb, 56
snowflake server, 4
Snowflake servers, 4
SSH
  hardening, 79
  sshd_config.d, 79
ssh
  debugging, 96
ssh config, 44
state
  absent, 13
string interpolation, 16
subscribes, 36
sudo, 44
symbolic link, 30
systemd, 33
  unit files, 73

T

tags, 57, 111
team workflow, 98
templates, 25, 111
  errors, 27
  locals, 26
  location, 26
  lookup, 57
TLS, 64

U

unattended upgrades, 79
unified diff, 11
unreachable, 95
user resource, 39, 105
  system, 73

V

validate
  --platform, 89
  limits, 89
vars, 25, 111
  from the command line, 27
  one place, 78
  precedence, 27, 50
vars.rb, 25
verify, 22, 80, 89
VPS, 98

W

workstation, 99