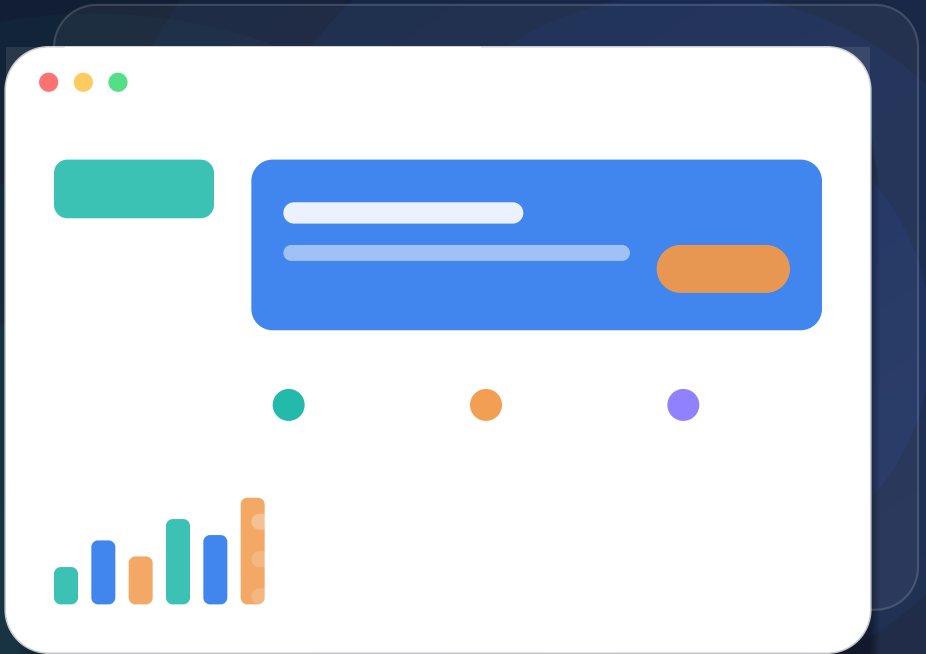


SSH AND APACHE WEB SERVER GUIDE

RUN A WEB SERVER REMOTELY



**A Practical Guide to
Running Apache over SSH**

CONNECT • CONFIGURE • SECURE • SERVE

SSH and Apache

A Practical Guide to Running a Web Server Remotely

This guide is provided for general information. Software versions, package names, file paths and defaults differ between Linux distributions and releases, so check your distribution documentation and test on a non-production server before changing a live one.

Contents

- Introduction 4
- 1. The Big Picture 6
- 2. Connecting and Preparing the Server 9
- 3. Installing Apache 13
- 4. Configuration Basics 16
- 5. Putting Your Website Online 20
- 6. Virtual Hosts: Many Sites, One Server 23
- 7. HTTPS and Certificates 27
- 8. Modules, .htaccess and URL Rewriting 32
- 9. PHP, Proxies and Applications 36
- 10. Securing the Server and Apache 40
- 11. Performance and Tuning 44
- 12. Logs, Monitoring and Troubleshooting 48
- 13. Deploying and Maintaining Your Server 53
- Appendix A: Quick Reference and Go-Live Checklist 57
- Appendix B: Glossary 59

Introduction

Most of the websites on the internet are served by a small piece of software running on a computer you will never see. That computer is usually in a data centre, rented by the hour, and managed entirely from a distance. The tools that make this possible are two old and trusted friends: **SSH**, which lets you log in to and control a remote computer securely, and **Apache**, one of the most widely used web servers in the world.

This book teaches you to use them together. You will learn how to connect to a server, install and configure Apache, host one or several websites, secure them with HTTPS, run PHP and other applications behind Apache, harden the server against attack, tune its performance, and find and fix problems when things go wrong. Everything is done the way professionals do it: from the command line, over SSH.

No previous server experience is required, but you should be comfortable opening a terminal and typing commands. Each step is explained, and each command is shown in full so that you can follow along on your own server.

What you will need

- A **Linux server** you can log in to. A small virtual private server (VPS) from any hosting provider is ideal, or a virtual machine on your own computer. Debian, Ubuntu, Rocky Linux, AlmaLinux and Fedora are all covered.
- An **SSH client**. Linux and macOS have one built in. Windows 10 and 11 include one in PowerShell and Windows Terminal.
- A **domain name** (optional, but needed for HTTPS with a public certificate).

- About an hour for the basics, and a willingness to experiment on a server you can afford to break.

How to use this book

- **New to servers?** Read Chapters 1 to 5 in order.
- **Hosting several sites?** Go to Chapter 6.
- **Ready for HTTPS?** Chapter 7 covers certificates and redirects.
- **Running PHP, Node.js or Python?** See Chapter 9.
- **Something is broken?** Chapter 12 decodes common errors.
- **Need a quick lookup?** Appendix A has paths, commands and a go-live checklist.

WARNING

A server on the internet is probed by automated attackers within minutes of going online. Practise on a throwaway server first, keep a second SSH session open whenever you change access settings, and never test risky changes on a server that holds data you cannot afford to lose.

TIP

This book is written for Apache HTTP Server 2.4, the current stable series. Commands for Debian and Ubuntu are shown first, with the equivalent for Red Hat-family systems (Rocky, AlmaLinux, Fedora) alongside. Package names, file paths and defaults can vary between releases, so check your distribution's documentation if something differs.

The Big Picture

Before typing commands, it helps to see how the pieces fit together. A web server project is small in concept: one remote computer, one program serving files, and you in control from a distance.

What a web server does

A **web server** is a program that waits for requests from browsers and sends back responses. When someone visits `http://example.com/about.html`, their browser asks the server for that page, and the server finds the file and returns it, together with a status code such as **200** (OK) or **404** (not found).

- For **static content** (HTML, CSS, images), the server simply reads files from disk.
- For **dynamic content** (PHP, Python, Node.js), the server hands the request to an application, which builds the page on the fly.

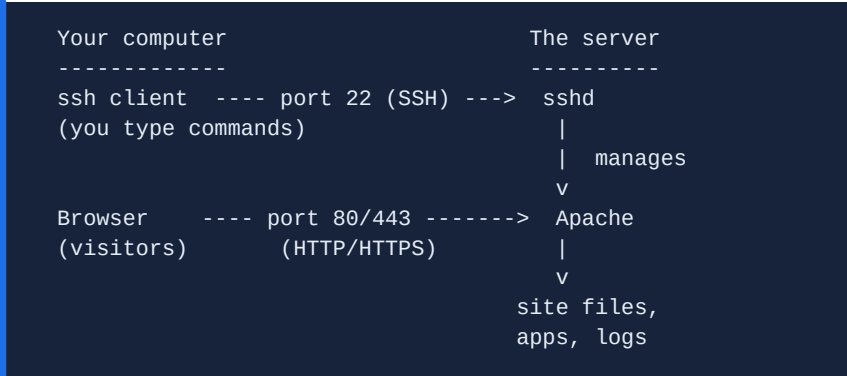
What Apache is

The **Apache HTTP Server**, often just called Apache or `httpd`, has been serving websites since 1995. It is open source, runs on nearly every operating system, and is known for its flexibility. Its features come from **modules**: small add-ons you switch on and off, such as URL rewriting, compression, HTTPS, proxying and authentication.

What SSH is

SSH (Secure Shell) gives you an encrypted command line on a remote computer. Passwords, commands and file contents all travel safely over it. It is how you install software, edit configuration, upload your website and read logs without ever sitting in front of the server.

How the pieces fit



Two different doors lead into the same machine. **Port 22** is for you, the administrator. **Ports 80 and 443** are for the public. A key part of server security is keeping the first door tightly locked while the second stays open.

Debian family or Red Hat family?

Linux distributions fall into two main groups when it comes to Apache. The software is the same, but names and locations differ. Learn both sets once and you can work on almost any server.

Item	Debian / Ubuntu	RHEL / Fedora / Rocky
Package	apache2	httpd
Service name	apache2	httpd
Main config	/etc/apache2/apache2.conf	/etc/httpd/conf/httpd.conf
Site configs	/etc/apache2/sites-available/	/etc/httpd/conf.d/
Document root	/var/www/html	/var/www/html
Runs as user	www-data	apache

Item	Debian / Ubuntu	RHEL / Fedora / Rocky
Logs	/var/log/apache2/	/var/log/httpd/
Enable a site	a2ensite NAME	Add a file to conf.d/
Enable a module	a2enmod NAME	Edit conf.modules.d/
Test config	apache2ctl configtest	apachectl configtest
Firewall tool	ufw	firewalld

TIP

Throughout the book, `apache2` means the service on Debian and Ubuntu, and `httpd` means the same on Red Hat-family systems. If you copy a command, swap the name to suit your system.

Connecting and Preparing the Server

A fresh server is usually delivered with a public IP address and a login for the **root** user. This chapter takes you from that first login to a safer, properly prepared machine. It covers only what you need for Apache; for a fuller treatment of SSH, see the companion *SSH Cheat Sheet*.

Your first login

```
$ ssh root@203.0.113.10
```

The first time, SSH shows the server's key fingerprint and asks you to confirm. Check it against the fingerprint in your hosting provider's console if it shows one, then type **yes**. Then enter the password or use the key your provider set up.

Update the system

```
# Debian / Ubuntu
$ apt update && apt upgrade -y

# RHEL / Fedora / Rocky
$ dnf upgrade -y
```

If the update installs a new kernel, reboot with `reboot` and reconnect after a minute.

Create a normal user with sudo rights

Working as root is risky, because every mistake has full power. Create a day-to-day user:

```
# Debian / Ubuntu
$ adduser deploy
$ usermod -aG sudo deploy

# RHEL / Fedora / Rocky
$ useradd -m deploy
$ passwd deploy
$ usermod -aG wheel deploy
```

Set up key-based login

On **your own computer**, create a key if you do not have one, and copy it to the server:

```
$ ssh-keygen -t ed25519 -C "you@example.com"
$ ssh-copy-id deploy@203.0.113.10
$ ssh deploy@203.0.113.10    # test: no password
```

Give the server a short name

Add an entry to `~/.ssh/config` on your computer so that you can type `ssh web` instead of the whole command:

```
Host web
  HostName 203.0.113.10
  User deploy
  IdentityFile ~/.ssh/id_ed25519
  ServerAliveInterval 60
```

Tighten SSH once keys work

Only after you have confirmed that key login works, edit `/etc/ssh/sshd_config` on the server and set:

```
PermitRootLogin no
PasswordAuthentication no
PubkeyAuthentication yes
```

```
$ sudo sshd -t                # check for errors
$ sudo systemctl reload ssh    # Debian / Ubuntu
$ sudo systemctl reload sshd   # RHEL family
```

WARNING

Keep your original SSH session open while you do this. Open a **second** terminal and confirm that you can still log in before closing the first. A mistake here can lock you out of a remote server.

Set up a basic firewall

A firewall lets in only the traffic you expect. Allow SSH **first**, then the web ports, and only then switch the firewall on.

```
# Debian / Ubuntu (ufw)
$ sudo ufw allow OpenSSH
$ sudo ufw allow 80/tcp
$ sudo ufw allow 443/tcp
$ sudo ufw enable
$ sudo ufw status

# RHEL / Fedora / Rocky (firewalld)
$ sudo firewall-cmd --permanent --add-service=http
$ sudo firewall-cmd --permanent --add-service=https
$ sudo firewall-cmd --reload
```

On RHEL-family systems, firewalld normally already allows SSH. Check with `sudo firewall-cmd --list-all`.

Handy habits for remote work

- Edit configuration with `sudoedit /etc/apache2/...` or `sudo nano`, whichever you prefer.
- Run long jobs inside **tmux** so a dropped connection does not kill them.
- Make a backup before editing any file: `sudo cp file file.bak`.

- Use `ssh web 'command'` to run a single command without a full login.

TIP

Check what the server looks like from outside: `ping 203.0.113.10` from your computer, and use `ssh -v web` if a login fails. The verbose output usually names the problem.

CHAPTER 3

Installing Apache

Installing Apache takes one command. Understanding what you have installed, and how to control it, takes a little longer, and is time well spent.

Install the package

```
# Debian / Ubuntu
$ sudo apt update
$ sudo apt install apache2

# RHEL / Fedora / Rocky
$ sudo dnf install httpd
```

On Debian and Ubuntu, Apache starts automatically. On Red Hat-family systems, start it and set it to run at every boot:

```
$ sudo systemctl enable --now httpd
```

Check that it is running

```
$ sudo systemctl status apache2      # or httpd
$ curl -I http://localhost
HTTP/1.1 200 OK
Server: Apache
```

Now visit `http://YOUR_SERVER_IP/` in a browser. You should see Apache's default welcome page. If it does not load, check the firewall (Chapter 2), your provider's network rules or security groups, and Chapter 12.

Controlling the service

Command	What it does
<code>sudo systemctl start apache2</code>	Start the server
<code>sudo systemctl stop apache2</code>	Stop it. The site goes offline
<code>sudo systemctl restart apache2</code>	Stop and start. Drops active connections
<code>sudo systemctl reload apache2</code>	Re-read configuration gracefully, with no downtime
<code>sudo systemctl enable apache2</code>	Start automatically at boot
<code>sudo systemctl status apache2</code>	Show whether it is running, with recent log lines
<code>sudo apache2ctl configtest</code>	Check configuration for errors before applying

TIP

Use **reload** rather than **restart** for configuration changes whenever possible. It applies changes without dropping visitors. Always run `configtest` first, so that a typo does not take the site down.

Where things live

On Debian and Ubuntu, the Apache configuration is split into tidy folders:

Path	Purpose
<code>/etc/apache2/apache2.conf</code>	The main configuration file
<code>/etc/apache2/ports.conf</code>	Which ports Apache listens on
<code>/etc/apache2/sites-available/</code>	Configuration for each website (all sites)
<code>/etc/apache2/sites-enabled/</code>	Links to the sites that are switched on

Path	Purpose
<code>/etc/apache2/mods-available/</code>	All installed modules and their settings
<code>/etc/apache2/mods-enabled/</code>	Links to the modules that are switched on
<code>/etc/apache2/conf-available/</code>	Extra configuration snippets
<code>/var/www/html/</code>	The default document root, where site files live
<code>/var/log/apache2/</code>	Access and error logs

On Red Hat-family systems, everything is simpler: the main file is `/etc/httpd/conf/httpd.conf`, and any file ending in `.conf` that you place in `/etc/httpd/conf.d/` is loaded automatically.

Useful inspection commands

```
$ apache2ctl -v          # Apache version
$ apache2ctl -V          # build details, including MPM
$ apache2ctl -M          # loaded modules
$ apache2ctl -S          # virtual host summary
$ sudo ss -tlnp | grep apache # listening ports
```

On Red Hat-family systems, use `apachectl` or `httpd -M` in place of `apache2ctl`.

Keep the server updated

Apache receives security fixes through your distribution's normal updates. Run updates regularly, and consider enabling automatic security updates (the `unattended-upgrades` package on Debian and Ubuntu, or `dnf-automatic` on Red Hat-family systems).

Configuration Basics

Apache is controlled by plain-text configuration files made of **directives**, which are simple instructions. Once you understand the structure, any configuration file becomes readable.

Directives and containers

A directive is a name followed by a value:

```
ServerName example.com
DocumentRoot /var/www/example.com/public
DirectoryIndex index.html index.php
```

Some directives apply only to certain places. These are wrapped in **containers**, which look like HTML tags:

Container	Applies to
<VirtualHost>	One website, defined by address or name
<Directory>	A folder on disk and everything inside it
<Files> and <FilesMatch>	Files by name or pattern
<Location>	A URL path, regardless of files on disk
<IfModule>	Only if a module is loaded

A first look at a site definition

```
<VirtualHost *:80>
    ServerName example.com
    ServerAdmin webmaster@example.com
    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks
        AllowOverride None
        Require all granted
    </Directory>
</VirtualHost>
```

The directives you will use most

Directive	Meaning
Listen 80	The port Apache accepts connections on
ServerName	The main hostname of the site
ServerAlias	Extra hostnames for the same site
DocumentRoot	The folder the site's files live in
DirectoryIndex	Which file to show for a folder, such as index.html
ErrorLog, CustomLog	Where errors and visits are recorded
Options	Features for a folder, such as Indexes or FollowSymLinks
AllowOverride	Whether .htaccess files may change settings
Require	Who may access the resource
ErrorDocument	A custom page shown for an error code
Include	Load another configuration file

Access control in Apache 2.4

Apache 2.4 uses the **Require** directive. The older **Order**, **Allow** and **Deny** directives are obsolete, and copying an old tutorial that uses them is a common source of errors.

```
Require all granted          # anyone
Require all denied          # no one
Require ip 203.0.113.0/24    # only this network
Require local                # only the server itself
Require valid-user           # any logged-in user
```

Safe editing workflow

1. **Back up** the file: `sudo cp site.conf site.conf.bak`.
2. **Edit** it with your editor of choice.
3. **Test**: `sudo apache2ctl configtest`. You want to see **Syntax OK**.
4. **Reload**: `sudo systemctl reload apache2`.
5. **Check** the result in a browser or with `curl`.
6. If anything breaks, **restore** the backup and reload.

Reading errors from configtest

```
$ sudo apache2ctl configtest
AH00526: Syntax error on line 7 of
/etc/apache2/sites-enabled/example.conf:
Invalid command 'RewriteEngine', perhaps misspelled
or defined by a module not included in the server
```

The message names the file and line. Here the fix is to enable the missing module (Chapter 8). A warning such as *Could not reliably determine the server's fully qualified domain name* is harmless, and goes away if you add `ServerName localhost` to the global configuration.

Where to put your own settings

- **Debian and Ubuntu:** put site settings in `sites-available/` and extra snippets in `conf-available/`, then enable them.
- **Red Hat family:** put files in `/etc/httpd/conf.d/`.
- Avoid editing the main file where you can. Your changes stay separate and survive package upgrades.

TIP

Comments start with `#`. Comment out a line instead of deleting it when you are experimenting, so you can easily put it back.

Putting Your Website Online

With Apache running, it is time to serve a real site. This chapter covers moving files onto the server with SSH tools, setting permissions correctly, and making sure the visitor sees what you intend.

Create a folder for the site

```
$ sudo mkdir -p /var/www/example.com/public
$ sudo chown -R deploy:www-data /var/www/example.com
$ sudo chmod -R 755 /var/www/example.com
```

On Red Hat-family systems, replace `www-data` with `apache`. Here `deploy` (you) owns the files, and the web server's group can read them.

Add a test page

```
$ cat > /var/www/example.com/public/index.html <<'EOF'
<!DOCTYPE html>
<html lang="en">
<head><title>Hello</title></head>
<body><h1>It works over SSH!</h1></body>
</html>
EOF
```

Uploading files with SSH tools

You rarely type pages directly on the server. Instead, you build them on your computer and upload them. Three SSH-based tools do this:

```
# scp: simple copy
$ scp index.html web:/var/www/example.com/public/
$ scp -r mysite/ web:/var/www/example.com/

# rsync: fast, copies only changes (best for deploys)
$ rsync -avz --delete mysite/ \
    web:/var/www/example.com/public/

# sftp: interactive
$ sftp web
```

WARNING

A trailing slash changes what rsync does. `mysite/` copies the *contents* of the folder, while `mysite` copies the folder itself. With `--delete`, a wrong path can erase files on the server. Add `-n` for a dry run first.

Getting permissions right

The web server runs as an unprivileged user (`www-data` or `apache`). It needs to **read** your files, but normally should not be able to **change** them. That way, a flaw in a web application cannot rewrite your site.

Item	Recommended setting
Folders	755 (owner can write; everyone can read and enter)
Files	644 (owner can write; everyone can read)
Owner	The deploy user, not the web server user
Folders the app must write to (uploads, cache)	775 with the web server group, or ownership by the web user
Secret files (configuration with passwords)	640, readable only by owner and the web group
Never	777 on anything

```
$ cd /var/www/example.com
$ sudo find . -type d -exec chmod 755 {} +
$ sudo find . -type f -exec chmod 644 {} +
```

Turn off directory listings

If a folder has no index file, Apache may list its contents, exposing files you did not mean to share. The option `-Indexes` in your `<Directory>` block prevents this.

Custom error pages

```
ErrorDocument 404 /errors/404.html
ErrorDocument 500 /errors/500.html
```

Friendly error pages keep visitors on your site and help them find their way back.

Test from outside

```
$ curl -I http://203.0.113.10/
$ curl -s http://203.0.113.10/ | head
```

Test from your own computer, not only from the server, so that you see what the public sees. Also try a phone on mobile data, which skips any office network rules.

TIP

Keep a simple deploy script on your computer: one `rsync` command in a file called `deploy.sh`. A repeatable one-step upload prevents mistakes, and is easy to hand to a colleague.

Virtual Hosts: Many Sites, One Server

A **virtual host** lets one Apache server host many websites. Each site has its own domain name, files, logs and settings, yet they share the same machine and IP address. Apache chooses the right one by reading the **Host** header that every browser sends.

Point your domain to the server

In your domain registrar or DNS provider's control panel, create **A records** (and **AAAA** records for IPv6) pointing to the server's IP address:

Name	Type	Value
example.com	A	203.0.113.10
www.example.com	A (or CNAME to example.com)	203.0.113.10

DNS changes can take from minutes to hours to spread. Check with `dig +short example.com`.

Create a virtual host (Debian and Ubuntu)

Create the file
`/etc/apache2/sites-available/example.com.conf`:

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    DocumentRoot /var/www/example.com/public

    <Directory /var/www/example.com/public>
        Options -Indexes +FollowSymLinks
        AllowOverride None
        Require all granted
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/example.com-error.log
    CustomLog \
        ${APACHE_LOG_DIR}/example.com-access.log combined
</VirtualHost>
```

Then enable it, test and reload:

```
$ sudo a2ensite example.com.conf
$ sudo a2dissite 000-default.conf # optional
$ sudo apache2ctl configtest
$ sudo systemctl reload apache2
```

The same on Red Hat-family systems

Save the file as `/etc/httpd/conf.d/example.com.conf` with the same contents, but replace the log lines with:

```
ErrorLog logs/example.com-error.log
CustomLog logs/example.com-access.log combined
```

Then run `sudo apachectl configtest` and `sudo systemctl reload httpd`.

Adding a second site

Repeat the process with another folder, another file and another `ServerName`. Both sites listen on port 80, and Apache sends each visitor to the right one.

The default virtual host

If a request arrives for a name that matches no virtual host, Apache serves the **first** one it loaded. Make sure that is a harmless placeholder page, not one of your real sites. On Debian and Ubuntu, `000-default.conf` loads first because of its name, so many administrators leave it as a simple placeholder.

Check what Apache sees

```
$ sudo apache2ctl -S
VirtualHost configuration:
*:80 is a NameVirtualHost
    default server example.com (...:1)
    port 80 namevhost example.com (...:1)
    port 80 namevhost another.org (...:1)
```

This is the best first step whenever the wrong site appears.

Testing before DNS is ready

You can test a new site before its DNS exists by telling `curl` which IP to use:

```
$ curl -I --resolve example.com:80:203.0.113.10 \
    http://example.com/
```

Or add a line to the hosts file on your computer (`/etc/hosts` on Linux and macOS) so that your browser uses the new server for that name.

Redirect www to non-www (or the reverse)

```
<VirtualHost *:80>
    ServerName www.example.com
    Redirect permanent / http://example.com/
</VirtualHost>
```

Pick one canonical address for each site, and redirect the other to it. This avoids duplicate content, and keeps search engines and cookies tidy.

Per-site logs and users

- Give each site its **own log files**, as above, so problems are easy to trace.
- For stronger isolation on shared servers, consider separate system users and PHP-FPM pools per site (Chapter 9).
- Keep each site's files in its own folder under `/var/www/`.

TIP

Name your configuration files after the domain, such as `example.com.conf`. When you have twenty sites, you will be glad you did.

HTTPS and Certificates

Today a website without HTTPS is treated as untrustworthy. Browsers mark plain HTTP sites as "Not secure", search engines favour HTTPS, and anything sent over HTTP, such as passwords or form data, can be read by anyone in between. Fortunately, free, automated certificates make HTTPS easy.

How HTTPS works, briefly

- A **certificate** proves that the server really controls the domain, and is signed by a trusted **certificate authority (CA)**.
- **TLS**, the protocol behind HTTPS, uses the certificate to set up an encrypted connection with the visitor.
- The **private key** stays on the server and must never be shared.

Free certificates with Let's Encrypt

Let's Encrypt is a free, automated certificate authority. A tool called **Certbot** requests the certificate, installs it in Apache and renews it automatically. Before you begin, make sure that:

- Your domain's **DNS points to the server** (Chapter 6).
- **Ports 80 and 443** are open in the firewall and in your provider's network rules.
- The site already works over **plain HTTP**, with a matching `ServerName`.

```
# Debian / Ubuntu
$ sudo apt install certbot python3-certbot-apache
$ sudo certbot --apache -d example.com -d www.example.com

# RHEL / Fedora / Rocky (EPEL may be needed)
$ sudo dnf install certbot python3-certbot-apache
$ sudo certbot --apache -d example.com -d www.example.com
```

Certbot asks for an email address (for expiry notices), checks that you control the domain, creates the certificate, edits your virtual host and offers to redirect HTTP to HTTPS. Choose the redirect. The installation method for Certbot changes between operating system releases, so also check the instructions on the Certbot website for your system.

Renewal

Let's Encrypt certificates last about 90 days. Certbot installs a timer that renews them automatically. Test it with a dry run:

```
$ sudo certbot renew --dry-run
$ systemctl list-timers | grep certbot
$ sudo certbot certificates          # list and expiry
```

What an HTTPS virtual host looks like

Certbot writes this for you, but it is worth understanding:

```
<VirtualHost *:443>
    ServerName example.com
    DocumentRoot /var/www/example.com/public

    SSLEngine on
    SSLCertificateFile \
        /etc/letsencrypt/live/example.com/fullchain.pem
    SSLCertificateKeyFile \
        /etc/letsencrypt/live/example.com/privkey.pem
</VirtualHost>
```

If you set it up by hand, enable the module first on Debian and Ubuntu: `sudo a2enmod ssl`.

Redirect HTTP to HTTPS

```
<VirtualHost *:80>
    ServerName example.com
    ServerAlias www.example.com
    Redirect permanent / https://example.com/
</VirtualHost>
```

Choose safe protocol settings

Modern Apache and OpenSSL defaults are sound, but you can make the intention explicit. Place this globally, or in each HTTPS virtual host:

```
SSLProtocol -all +TLSv1.2 +TLSv1.3
SSLHonorCipherOrder off
```

This turns off old, weak protocol versions. For a complete, current list of recommended settings, use a reputable configuration generator and your distribution's defaults rather than inventing your own cipher list.

HSTS: telling browsers to insist on HTTPS

HTTP Strict Transport Security tells browsers to use only HTTPS for your domain for a set time. It needs the `headers` module.

```
Header always set Strict-Transport-Security \
    "max-age=31536000"
```

WARNING

HSTS is hard to undo. Once a browser has seen the header, it will refuse plain HTTP for the full period. Start with a short value, such as a day, and raise it only when you are sure HTTPS works everywhere on the site.

Turn on HTTP/2

HTTP/2 makes pages load faster over HTTPS. Enable the module and add a protocols line (it works best with the **event** MPM, see Chapter 11):

```
$ sudo a2enmod http2

# in the HTTPS virtual host
Protocols h2 http/1.1
```

Check your setup

```
$ curl -I https://example.com/
$ openssl s_client -connect example.com:443 \
    -servername example.com < /dev/null | head -20
$ echo | openssl s_client -connect example.com:443 \
    -servername example.com 2>/dev/null | \
    openssl x509 -noout -dates
```

The last command shows when the certificate starts and expires. Online tools such as the Qualys SSL Labs test grade your configuration in detail.

Certificates without a public domain

For internal servers and testing, you can create a **self-signed** certificate with `openssl`. Browsers will warn that it is untrusted, so use it only for private experiments. For private networks that need trusted certificates, use an internal certificate authority.

TIP

If Certbot fails, the cause is nearly always DNS (the domain does not point to the server yet), a closed port 80, or a mismatched `ServerName`. Check those three first.

Modules, .htaccess and URL Rewriting

Apache's power lies in its modules. Switching on a module is how you gain features such as redirects, compression, security headers and proxying.

Managing modules

```
$ apache2ctl -M          # list loaded modules
$ sudo a2enmod rewrite headers # Debian / Ubuntu: enable
$ sudo a2dismod autoindex  # disable
$ sudo systemctl restart apache2
```

Enabling or disabling a module needs a **restart**, not just a reload. On Red Hat-family systems, modules are loaded by LoadModule lines in /etc/httpd/conf.modules.d/.

Modules worth knowing

Module	What it provides
mod_rewrite	Powerful URL rewriting and redirects
mod_headers	Add or change HTTP headers, such as security headers
mod_ssl	HTTPS and TLS
mod_deflate	Compress responses to save bandwidth
mod_expires	Tell browsers how long to cache files
mod_http2	The HTTP/2 protocol
mod_proxy and friends	Forward requests to other servers or applications

Module	What it provides
<code>mod_proxy_fcgi</code>	Talk to PHP-FPM and other FastCGI backends
<code>mod_auth_basic</code>	Simple username and password protection
<code>mod_status</code>	A live status page about the server
<code>mod_remoteip</code>	Recover real visitor addresses behind a proxy or CDN

.htaccess files

An `.htaccess` file lets you change settings for a folder without editing the main configuration, which is handy on shared hosting and for applications that ship their own rules. It only works if `AllowOverride` permits it for that folder.

Approach	Pros	Cons
Settings in the main configuration (virtual host)	Faster, since read once at start. More secure and easier to audit	Needs a reload and server access
<code>.htaccess</code> files	Changes apply instantly. Works without server access	Apache must check for them on every request, which slows things. Scattered rules are hard to audit

If you control the server, prefer putting rules in the virtual host and setting `AllowOverride None`. Enable `.htaccess` only where an application needs it, for example `AllowOverride All` for a content management system folder.

URL rewriting basics

`mod_rewrite` changes URLs based on rules. Rules are written as conditions and a rewrite rule, which uses a pattern (a regular expression).

Force HTTPS (when you do not use Certbot's redirect):

```
RewriteEngine On
RewriteCond %{HTTPS} off
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
```

Send every request to one script (a "front controller", used by many frameworks and CMSs):

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^ index.php [L]
```

A simple redirect for a moved page:

```
Redirect permanent /old-page.html /new-page.html
```

Flag	Meaning
[L]	Last rule: stop processing further rules
[R=301]	Send a permanent redirect to the browser
[R=302]	Send a temporary redirect
[NC]	Case-insensitive match
[QSA]	Keep the query string when rewriting

TIP

For a plain redirect, use `Redirect` or `RedirectMatch`. They are simpler and clearer than rewrite rules. Reach for `mod_rewrite` only when you need conditions.

Compression and caching

```
# mod_deflate
AddOutputFilterByType DEFLATE text/html text/css \
    application/javascript application/json image/svg+xml

# mod_expires
ExpiresActive On
ExpiresByType image/png "access plus 1 month"
ExpiresByType text/css "access plus 1 week"
```

Password-protect a folder

For a quick, simple lock on a private area, use basic authentication. It is not a substitute for a real login system, and should only be used over HTTPS, since the password is sent with every request.

```
$ sudo apt install apache2-utils      # htpasswd tool
$ sudo htpasswd -c /etc/apache2/.htpasswd alice
$ sudo htpasswd /etc/apache2/.htpasswd bob    # add more
```

```
<Directory /var/www/example.com/public/private>
    AuthType Basic
    AuthName "Restricted area"
    AuthUserFile /etc/apache2/.htpasswd
    Require valid-user
</Directory>
```

Keep the password file **outside** the document root, so it can never be downloaded. On Red Hat-family systems, the tool comes from the httpd-tools package.

PHP, Proxies and Applications

Static pages are only the start. Most real sites run an application: a content management system written in PHP, a service built with Node.js or Python, or a mix. Apache is happy to host them all, either by running the code itself or by passing requests to another program.

Running PHP

There are two main ways to run PHP with Apache:

Method	Description
mod_php	PHP runs inside Apache. Simple, but forces the older prefork MPM and uses more memory
PHP-FPM (recommended)	PHP runs as a separate service. Apache hands requests to it. Works with the faster event MPM and allows per-site isolation

To set up PHP-FPM on Debian and Ubuntu:

```
$ sudo apt install php-fpm
$ sudo a2enmod proxy_fcgi setenvif
$ sudo a2enconf php8.3-fpm      # use your PHP version
$ sudo systemctl reload apache2
```

The version number in the last configuration name depends on the PHP version installed (check with `ls /etc/apache2/conf-available/ | grep fpm`). Create a test page and visit it, then delete it:

```
$ echo '<?php phpinfo();' | \  
    sudo tee /var/www/example.com/public/info.php  
$ sudo rm /var/www/example.com/public/info.php
```

WARNING

Never leave a `phpinfo()` page on a live server. It reveals detailed information that helps attackers.

On Red Hat-family systems, install `php-fpm`, enable it with `sudo systemctl enable --now php-fpm`, and Apache's default configuration normally passes `.php` files to it already.

A database for your application

Many PHP applications need MariaDB or MySQL. Install it, run the security script, and create a database and a dedicated user with limited rights for each application:

```
$ sudo apt install mariadb-server  
$ sudo mysql_secure_installation
```

Keep the database listening on the local machine only, and never expose its port to the internet. If you need to reach it from your own computer, use an **SSH tunnel** instead:

```
$ ssh -L 3307:127.0.0.1:3306 web  
# then connect your database tool to localhost:3307
```

Reverse proxy to another application

Applications written in Node.js, Python, Ruby or Go usually run their own small web server on a local port, such as 3000. Apache can sit in front as a **reverse proxy**, handling HTTPS, compression and logging, while forwarding requests to the application.

```
$ sudo a2enmod proxy proxy_http headers
$ sudo systemctl restart apache2
```

```
<VirtualHost *:443>
    ServerName app.example.com
    # ... SSL settings from Chapter 7 ...

    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:3000/
    ProxyPassReverse / http://127.0.0.1:3000/

    RequestHeader set X-Forwarded-Proto "https"
</VirtualHost>
```

For applications that use WebSockets, also enable `proxy_wstunnel` and proxy the WebSocket path with a `ws://` address.

WARNING

Never set `ProxyRequests On` on a public server. That turns Apache into an **open forward proxy**, which attackers use to hide their traffic. A reverse proxy needs only `ProxyPass`, with `ProxyRequests` left at its default of `Off`.

Keeping applications running

A backend application should start at boot and restart if it crashes. The standard tool is a **systemd service**. A small unit file in `/etc/systemd/system/myapp.service` does the job:

```
[Unit]
Description=My application
After=network.target

[Service]
User=deploy
WorkingDirectory=/var/www/myapp
ExecStart=/usr/bin/node server.js
Restart=on-failure

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

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable --now myapp
$ sudo journalctl -u myapp -f      # follow its logs
```

SELinux on Red Hat-family systems

These systems run **SELinux**, which stops Apache from doing things it should not, such as connecting to other services or writing to arbitrary folders. If a proxy or upload fails with no obvious reason, SELinux is a common cause. Do not disable it. Adjust it:

```
# allow Apache to connect to backends
$ sudo setsebool -P httpd_can_network_connect 1

# allow writes to an uploads folder
$ sudo semanage fcontext -a -t httpd_sys_rw_content_t \
    "/var/www/app/uploads(/.*)?"
$ sudo restorecon -Rv /var/www/app/uploads
```

TIP

Check `sudo ausearch -m avc -ts recent` to see recent SELinux denials. They name exactly what was blocked.

Securing the Server and Apache

Every server connected to the internet is attacked constantly. Most attacks are automated and look for easy targets: old software, weak passwords and exposed files. A handful of habits stops almost all of them.

Secure SSH, the front door to the machine

- Use **keys**, not passwords, and turn off password login (Chapter 2).
- Disable **root login**, and allow only specific users with `AllowUsers` if appropriate.
- Keep the SSH server **updated**.
- Limit SSH access by **firewall** to known addresses where practical.
- Install **fail2ban**, which temporarily blocks addresses that fail to log in repeatedly.
- Protect your private key with a **passphrase**.

```
$ sudo apt install fail2ban
$ sudo systemctl enable --now fail2ban
$ sudo fail2ban-client status sshd
```

Reduce what Apache reveals

By default, Apache can announce its exact version and operating system in headers and error pages. That helps attackers pick exploits. Turn it down:

```
ServerTokens Prod
ServerSignature Off
TraceEnable Off
```

On Debian and Ubuntu, these settings live in `/etc/apache2/conf-available/security.conf`. On Red Hat-family systems, add them to a file in `conf.d/`.

Add security headers

```
Header always set X-Content-Type-Options "nosniff"
Header always set X-Frame-Options "SAMEORIGIN"
Header always set Referrer-Policy \
    "strict-origin-when-cross-origin"
Header always set Permissions-Policy "geolocation=()"
```

A **Content-Security-Policy** header gives strong protection against script injection, but must be written carefully for each site, so test it thoroughly before relying on it.

Protect sensitive files

Version control folders, configuration backups and hidden files must never be served:

```
<DirectoryMatch "/\.(git|svn|hg)">
    Require all denied
</DirectoryMatch>

<FilesMatch "(^\.|\.(bak|old|sql|log|env)$)">
    Require all denied
</FilesMatch>
```

Keep configuration files with passwords **outside** the document root wherever possible.

Only run what you need

- **Disable unused modules** with `a2dismod`. Fewer modules mean fewer risks.
- **Remove default pages** and sample files.
- **Stop unused services** on the server, and close their ports.
- **Delete old sites** and test files rather than leaving them online.

Limit who can reach sensitive areas

```
<Location "/admin">  
    Require ip 203.0.113.0/24  
</Location>
```

Restricting an admin area to known addresses, or putting it behind an SSH tunnel and a local-only rule, removes it from the reach of most attackers.

Keep everything updated

- Apply operating system updates regularly, and enable automatic security updates.
- Update your **applications** (such as WordPress and its plugins) promptly. Outdated applications are the most common way sites are compromised.
- Remove plugins and themes you do not use.

Additional protection

Measure	Notes
Web application firewall (ModSecurity)	Inspects requests for attack patterns. Needs tuning to avoid blocking legitimate traffic
Fail2ban Apache jails	Ban addresses that probe for logins or bad bots, using Apache logs

Measure	Notes
Rate limiting	Slow down abusive clients using modules or a front-end service
CDN or reverse proxy service	Hides your server address and absorbs floods of traffic
Intrusion alerts and monitoring	Alert on unexpected logins and changes

Backups

Security includes recovery. Back up your site files, Apache configuration and databases regularly, store copies **off the server**, and practise restoring them. A backup you have never tested is only a hope.

```
# a nightly archive, pulled to your own computer
$ ssh web 'sudo tar czf - /etc/apache2 /var/www' \
  > backup-$(date +%F).tar.gz
```

TIP

Run `sudo ss -tlnp` on your server and look at every listening port. Anything you cannot explain should be investigated, and anything unnecessary should be closed.

Performance and Tuning

A well-built Apache server on modest hardware can handle a surprising amount of traffic. Good performance comes from simple choices made early, not from exotic tricks. Measure first, change one thing at a time, and measure again.

Choose the right MPM

The **Multi-Processing Module (MPM)** decides how Apache handles many connections at once. Apache offers three:

MPM	How it works	Best for
prefork	One process per connection. Simple and compatible, but memory-hungry	Older setups using mod_php
worker	Several processes, each with multiple threads	General use where thread safety is fine
event	Like worker, but handles idle keep-alive connections more efficiently	Modern sites, HTTP/2, PHP-FPM. The usual default

```
$ apache2ctl -V | grep -i mpm
Server MPM:      event

# switch on Debian / Ubuntu
$ sudo a2dismod mpm_prefork
$ sudo a2enmod mpm_event
$ sudo systemctl restart apache2
```

Use the **event** MPM with PHP-FPM wherever you can.

Keep-alive and timeouts

```
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 3
Timeout 60
```

KeepAlive lets a browser reuse one connection for many files, which speeds up pages. A short **KeepAliveTimeout** stops idle visitors from tying up resources.

Size the server's capacity

The most important limit is **MaxRequestWorkers**, the number of requests Apache will handle at the same moment. If it is too low, visitors queue up. If it is too high, the server runs out of memory and starts swapping, which is far worse.

1. Find the average memory used by one Apache process (check with **ps** or **top**).
2. Work out how much memory you can give to Apache, leaving room for the operating system, database and other services.
3. Divide, and set **MaxRequestWorkers** a little below the result.
4. Watch memory under load and adjust.

Serve less, and serve it smaller

- **Compress text** with **mod_deflate** (Chapter 8).
- **Set cache headers** so browsers keep images, scripts and stylesheets.
- **Optimise images**: use the right size and modern formats.
- **Combine and minimise** scripts and stylesheets where it helps.
- **Use a CDN** for static files if your visitors are spread around the world.

- **Cache dynamic pages** in the application or with a caching layer, since generating pages is far slower than serving files.

Watch the server live with mod_status

The mod_status module shows what Apache is doing right now: busy and idle workers, requests per second and uptime. On Debian and Ubuntu it is enabled and limited to the local machine by default:

```
<Location "/server-status">
    SetHandler server-status
    Require local
</Location>
```

Because it only answers requests from the server itself, you read it through an **SSH tunnel**, which is a neat use of everything you have learned:

```
$ ssh -L 8080:localhost:80 web

# in another terminal on your own computer
$ curl http://localhost:8080/server-status?auto
```

The status page is never something to expose to the public, because it reveals details about traffic and the server.

Measure before and after

```
$ curl -o /dev/null -s -w "%{time_total}s\n" \
    https://example.com/

$ ab -n 500 -c 20 https://example.com/
```

ab (ApacheBench, from apache2-utils) sends many requests and reports how the server copes. Only test servers you own, and never run it against a live production site at peak time.

Log rotation

Logs grow without limit unless they are rotated. Debian and Ubuntu install a rotation rule for Apache automatically (/etc/logrotate.d/apache2). Check that it exists, and that your own extra logs are covered. A full disk is a classic cause of mysterious failures.

```
$ df -h / # free disk space
$ sudo du -sh /var/log/apache2 # log size
```

TIP

Most slow sites are slow because of the application or database, not Apache. Before tuning Apache, find out where the time is actually spent.

Logs, Monitoring and Troubleshooting

When something goes wrong, the answer is nearly always in a log. Learning to read them calmly is the single most valuable administration skill.

Where the logs are

Log	Debian / Ubuntu	RHEL family
Access log	/var/log/apache2/access.log	/var/log/httpd/access_log
Error log	/var/log/apache2/error.log	/var/log/httpd/error_log
Service log	journalctl -u apache2	journalctl -u httpd
Login attempts	/var/log/auth.log	/var/log/secure

Read logs live

```
$ sudo tail -f /var/log/apache2/error.log
$ sudo tail -f /var/log/apache2/access.log
$ sudo journalctl -u apache2 -n 50 --no-pager
```

Run `tail -f` in one terminal while you reproduce the problem in your browser. You will see the exact message appear.

What an access log line contains

```
203.0.113.5 - - [18/Oct/2026:15:02:11 +0200]
"GET /about.html HTTP/1.1" 200 5123
"https://example.com/" "Mozilla/5.0 ..."
```

From left to right: the visitor's address, the time, the request, the status code, the size in bytes, the referring page and the browser.

Useful log questions

```
# busiest visitor addresses
$ sudo awk '{print $1}' access.log | sort | uniq -c \
  | sort -rn | head

# count of each status code
$ sudo awk '{print $9}' access.log | sort | uniq -c \
  | sort -rn

# most requested pages
$ sudo awk '{print $7}' access.log | sort | uniq -c \
  | sort -rn | head

# all 404 errors
$ sudo grep ' 404 ' access.log | tail
```

Common HTTP status codes

Code	Meaning	Usual cause
200	OK	Everything worked
301 / 302	Redirect	Moved page or HTTPS redirect
403	Forbidden	Permissions, missing Require, or SELinux
404	Not found	Wrong URL, wrong DocumentRoot or missing file
500	Server error	Application error or bad .htaccess. Check the error log
502	Bad gateway	The backend application or PHP-FPM is down
503	Service unavailable	Overloaded, or backend not ready

Code	Meaning	Usual cause
504	Gateway timeout	The backend took too long to respond

Common errors and what to do

Message or symptom	Likely cause and fix
Site does not load at all	Apache stopped, a firewall blocks 80 or 443, or DNS points elsewhere. Check systemctl status and the firewall
Address already in use (AH00072)	Another program uses port 80 or 443. Find it with <code>sudo ss -tlnp grep ':80 '</code>
AH00558: Could not reliably determine the server's fully qualified domain name	A harmless warning. Add ServerName localhost to the main configuration
Syntax error on line N	Typo in a configuration file. Run configtest to see the file and line
Invalid command 'RewriteEngine'	Module not enabled. Run <code>sudo a2enmod rewrite</code> and restart
403 Forbidden	Folder permissions, missing 'Require all granted', directory with no index file, or SELinux
AH01630: client denied by server configuration	A Require rule is refusing the visitor
Wrong website appears	ServerName mismatch or default virtual host. Check <code>apache2ctl -S</code>
Too many redirects	A redirect loop, often HTTPS redirects behind a proxy or CDN. Check rules and X-Forwarded-Proto
Certificate warning in browser	Expired or mismatched certificate. Run <code>certbot</code> certificates and renew
502 or 503 when proxying	The backend is not running or is on a different port. Check the app's service and logs

Message or symptom	Likely cause and fix
Page shows PHP source code	PHP is not connected to Apache. Check PHP-FPM and the handler configuration
Changes do not appear	Browser or CDN cache, or the config was not reloaded

A calm troubleshooting method

1. **Define the problem.** What exactly fails, for whom, since when? What changed just before?
2. **Is Apache running?** `systemctl status apache2`.
3. **Is the configuration valid?** `apache2ctl configtest`.
4. **Is it listening?** `sudo ss -tlnp | grep -E ':(80|443)'`.
5. **Can you reach it?** Test from the server (`curl -I localhost`), then from outside. If the first works and the second does not, suspect a firewall or DNS.
6. **Read the error log** while repeating the problem.
7. **Turn up the detail**, if needed, with `LogLevel debug`. Reload, test, then set it back to `warn`.
8. **Change one thing at a time**, and undo it if it does not help.

Simple monitoring

- Use an **uptime monitoring** service that checks your site every few minutes and alerts you.
- Watch **disk space**, **memory** and **certificate expiry**.
- Review **logs** weekly for strange traffic, errors and failed logins.
- Set up **email alerts** for failed backups and expiring certificates.

TIP

When you ask for help online, include the output of `apache2ctl -V` | `head`, the relevant configuration, and the last lines of the error log. A good question with good evidence is often answered in minutes.

Deploying and Maintaining Your Server

A server is not finished when the website appears. Healthy servers are maintained with simple routines that keep them safe, current and recoverable. This chapter ties together everything in the book into a routine you can live with.

A simple, repeatable deploy

A good deployment is boring: one command, the same every time, easy to undo. Three common patterns, from simplest to most robust:

1. **rsync from your computer.** Build the site locally and upload only the changes.
2. **git pull on the server.** The server holds a clone of your repository, and you update it over SSH with `ssh web 'cd /var/www/site && git pull'`. Use a read-only **deploy key** for the repository.
3. **Release folders with a symlink.** Upload each version to a new dated folder, then switch a current link to it. Rolling back is a single command.

```
# release folders and an instant rollback
$ rsync -az build/ web:/var/www/site/releases/2026-10-18/
$ ssh web 'ln -sf /var/www/site/releases/2026-10-18 \
/var/www/site/current'
# DocumentRoot points to /var/www/site/current
```

Try changes safely first

- Keep a **staging** copy of the site, on a separate virtual host or server, and test changes there first.
- Reach a private staging site through an **SSH tunnel** rather than exposing it publicly.
- Always run `configtest` before reloading.
- Make changes at quiet times, and keep a second SSH session open.

Routine maintenance

How often	Task
Daily or automatic	Security updates, backups, certificate renewal, uptime checks
Weekly	Review error logs and failed logins; check disk space and memory
Monthly	Apply all updates and reboot if needed; review users and SSH keys; test a backup restore
Quarterly	Review firewall rules, open ports and installed modules; review application plugins
Yearly	Renew domain names and hosting; review the whole design and your recovery plan

Scheduling jobs with cron

Use **cron** for repeating tasks such as backups. Edit your schedule with `crontab -e`:

```
# every night at 02:30, make a backup
30 2 * * * /home/deploy/bin/backup.sh \
>> /var/log/backup.log 2>&1
```

Managing access over time

- Give each person **their own account and key**. Never share a login.
- **Remove** keys from `~/.ssh/authorized_keys` as soon as someone no longer needs access.
- Use `sudo` rather than a shared root login, so actions are logged against a name.
- Review who can log in at least once a quarter.

A disaster recovery plan

Ask: if this server vanished tonight, how fast could I be back online? A good answer is a short written list:

1. Where the **backups** are, and how to retrieve them.
2. How to create a **new server** and get SSH access.
3. The commands to **install Apache** and any extras (keep them in a script).
4. How to **restore** configuration, site files and databases.
5. How to **point DNS** to the new address.
6. How to **reissue certificates**.

Better still, write your setup as a script or use configuration management tools (such as Ansible), so that a whole server can be rebuilt with a single command. Doing this once, even in a rough form, makes you far more confident when something fails.

Where to go next

- **Nginx** is another popular web server, often used in front of Apache or on its own. Learning both helps.
- **Containers** (Docker) package applications to run the same everywhere.

- **Configuration management** and **infrastructure as code** make servers repeatable.
- **Monitoring and logging stacks** help once you manage more than one server.
- The official **Apache HTTP Server documentation** is detailed and authoritative, and your distribution's wiki shows its specific conventions.

TIP

The best server administrators are not the ones who never have problems. They are the ones who keep backups, take notes, change one thing at a time, and always know how to undo what they just did.

Appendix A: Quick Reference and Go-Live Checklist

Everyday commands

Command	What it does
<code>ssh web</code>	Log in to the server (with your config alias)
<code>sudo systemctl reload apache2</code>	Apply config changes without downtime
<code>sudo apache2ctl configtest</code>	Check configuration syntax
<code>sudo apache2ctl -S</code>	Show virtual hosts and the default site
<code>sudo apache2ctl -M</code>	List loaded modules
<code>sudo a2ensite NAME.conf</code>	Enable a site (Debian / Ubuntu)
<code>sudo a2dissite NAME.conf</code>	Disable a site
<code>sudo a2enmod NAME</code>	Enable a module (then restart)
<code>sudo certbot renew --dry-run</code>	Test certificate renewal
<code>sudo tail -f error.log</code>	Follow the error log live
<code>curl -I https://example.com/</code>	Check the response headers
<code>sudo ss -tlnp</code>	List listening ports and their programs
<code>rsync -avzn src/ web:/dest/</code>	Dry run of an upload
<code>ssh -L 8080:localhost:80 web</code>	Tunnel to the server's port 80

Go-live checklist

1. **SSH is locked down:** keys only, no root login, a second session tested.
2. **The system is updated** and automatic security updates are on.
3. **The firewall** allows only SSH, 80 and 443.
4. **DNS** points to the server, and www and non-www are handled.
5. **Virtual host** uses the right ServerName, DocumentRoot and logs.
6. **Permissions** are correct: files 644, folders 755, nothing 777.
7. **Directory listing** is off, and default and test pages are removed.
8. **HTTPS** works, HTTP redirects to it, and renewal is tested.
9. **ServerTokens** and **ServerSignature** are reduced.
10. **Security headers** are set, and sensitive files are blocked.
11. **Unused modules** are disabled.
12. **The application** is updated, and its admin area is restricted.
13. **Compression and caching** are enabled.
14. **Logs** are rotated and disk space is monitored.
15. **Backups** run automatically, are stored off the server and have been tested.
16. **Uptime monitoring** and alerts are active.
17. **A recovery plan** is written down.

Appendix B: Glossary

A record. A DNS record that points a name to an IPv4 address.

Apache (httpd). The Apache HTTP Server, a widely used open-source web server.

Certificate. A file that proves a server's identity and enables HTTPS.

CGI / FastCGI. Ways for a web server to run external programs to generate pages.

Certbot. A tool that obtains and renews Let's Encrypt certificates.

Directive. A single configuration instruction in Apache.

DocumentRoot. The folder from which a website's files are served.

DNS. The system that turns domain names into IP addresses.

Firewall. Software that allows or blocks network traffic by rules.

HSTS. A header that tells browsers to use only HTTPS for a site.

.htaccess. A per-folder file of Apache settings.

HTTP / HTTPS. The web protocol, and its encrypted version.

Let's Encrypt. A free, automated certificate authority.

Module. An Apache add-on providing a feature, such as rewriting or SSL.

MPM. Multi-Processing Module: how Apache handles concurrent connections.

PHP-FPM. A separate service that runs PHP code for the web server.

Port. A numbered channel on a computer, such as 22 for SSH and 80 for HTTP.

Reverse proxy. A server that forwards visitors' requests to another application behind it.

SELinux. A Linux security system that limits what programs may do.

SSH. Secure Shell: an encrypted way to control a remote computer.

SSH key. A pair of files, one private and one public, used to log in without a password.

systemd. The service manager used by most modern Linux systems.

TLS. The protocol that encrypts HTTPS connections.

Tunnel. An encrypted path through SSH that carries other network traffic.

Virtual host. A configuration that lets one Apache server host many websites.