WEB SERVERS

Web Servers Fundamentals

Apache · Nginx · IIS

Architecture & Configuration Philosophy

Virtual Hosts, TLS, Reverse Proxying & Performance

Capstone: Choosing the Right Server for the Job

Philip Osztromok

Generated with Claude

Table of Contents

Web Servers Fundamentals — 10 Chapters

CHAPTER	TITLE
Chapter 1	**What a Web Server Actually Does**
Chapter 2	**Architecture Models**
Chapter 3	**Configuration Philosophy**
Chapter 4	**Virtual Hosts, Server Blocks & Sites**
Chapter 5	**Serving Static Content**
Chapter 6	**Reverse Proxying & Load Balancing Fundamentals**
Chapter 7	**TLS/HTTPS Termination at the Web Server Layer**
Chapter 8	**Logging, Access Control & Basic Hardening**
Chapter 9	**Performance Basics**
Chapter 10	**Capstone: Choosing and Configuring the Right Web Server for a Given Job**

Apache has only ever shown up as an implementation detail inside Setting Up a Web Server on Debian, and Nginx has had no coverage at all — despite both being some of the most widely used infrastructure software in existence. This course compares Apache, Nginx, and IIS directly: their architecture philosophies, their configuration models, and when each one is actually the right choice. This chapter starts with the one question underneath all three: what does a web server actually do?

What "Serving a Website" Actually Means

A web browser opens a TCP connection to a server and sends an HTTP request — a plain-text message naming a method (`GET`, `POST`, ...), a path, and a set of headers. A web server is the piece of software listening on that port, waiting for exactly this kind of message, that parses the request and sends back an HTTP response: a status code, response headers, and a body — the actual HTML, image, or data the browser asked for. Every single feature this course covers is built on top of this one request-in, response-out cycle.

Where a Web Server Sits in the Stack

A request typically passes through more than one piece of software before a response comes back: the browser talks to the web server, and the web server either answers directly (for a static file already sitting on disk) or hands the request off to something else — a PHP interpreter, a Python or Node.js application process, a database-backed application server — and then relays that result back to the browser as the final response. The web server's own specific job is narrower than "running the website": it's the thing standing at the front door, deciding what to do with each request that arrives.

The Three Servers This Course Compares

Apache HTTP Server (1995) is the oldest of the three, built around a module system and, historically, a process-per-connection model. **Nginx** (2004) was built specifically to handle far more simultaneous connections than Apache's original model could, using a fundamentally different event-driven architecture. **Internet Information Services (IIS)** is Microsoft's own web server, tightly integrated with Windows and the .NET ecosystem, configured through a GUI and XML rather than the plain-text config files the other two use. Chapter 2 goes into each of these architecture models directly — this chapter is only introducing the three names.

Static vs. Dynamic Content

Serving a static file — an image, a stylesheet, a plain HTML page — is a web server's own native job: read the file from disk, send it back, done. Dynamic content is different: a page that needs to run code, query a database, or generate different output per request has to be handed off to something that can actually execute that code, with the web server acting as the front-facing layer that receives the request and relays the eventual response. This distinction — what the web server does itself vs. what it hands off — comes up constantly throughout this course, especially once Chapter 6 covers reverse proxying.

	APACHE	NGINX	IIS
First released	1995	2004	1995 (bundled with Windows NT)
Platform	Cross-platform	Cross-platform	Windows only
Config format	Plain-text directives	Plain-text directives	GUI / XML
Best known for	Flexibility, module ecosystem	High-concurrency performance	Windows/.NET integration

CONNECTING THIS TO A REAL DEPLOYMENT YOU'VE ALREADY DONE

Setting Up a Web Server on Debian walked through installing and configuring a real Apache server on a real machine. Everything that course had you actually type and configure — virtual hosts, config directives, restarting the service — is Apache's own concrete implementation of the general request/response model this chapter just introduced. This course revisits that same territory from a comparative angle: not just "how do you configure Apache," but "why does Apache's own approach look the way it does, compared to Nginx's or IIS's."

A WEB SERVER IS NOT THE SAME THING AS A WEB APPLICATION

It's easy to blur "the web server" and "the website" together into one idea, especially when a single Apache install feels like it's "running" an entire site. The web server itself is only the layer handling connections and HTTP requests/responses — the actual application logic (a PHP script, a Django app, a Node.js process) is a separate piece of software the web server talks to, not something the web server is doing itself. Keeping this distinction clear is what makes the rest of this course's architecture comparisons make sense.

Hands-On Exercises

EXERCISE 1

In your own words, describe the full request/response cycle for a browser loading a plain HTML page from a web server — name every step between clicking a link and the page appearing.

View solution

EXERCISE 2

A page on a site displays a user's account balance, pulled live from a database. Explain which part of serving this page is the web server's own job, and which part has to be handed off to something else, and why.

View solution

EXERCISE 3

A friend says "my website runs on Apache" and separately "my website is built with Django." Explain, using this chapter's own distinction, why both statements can be true about the same site without contradicting each other.

View solution

CHAPTER 1 QUICK REFERENCE

- A web server's core job: accept HTTP requests, send back HTTP responses
- **Static content** — served directly from disk by the web server itself
- **Dynamic content** — handed off to a separate application process, with the web server relaying the result
- This course compares **Apache** (1995, modular), **Nginx** (2004, event-driven), and **IIS** (Windows/.NET-integrated)
- A web server and a web application are two different pieces of software working together, not one and the same

Chapter 1 established what a web server does in the abstract: accept connections, parse requests, send back responses. This chapter is about how each of the three servers actually handles doing that many times at once — the specific architectural choice behind each one, and why it matters once real traffic shows up.

Apache's Multi-Processing Modules (MPMs)

Apache handles concurrency through a pluggable **Multi-Processing Module**, chosen at configuration time rather than fixed. **Prefork** is the oldest and simplest: one entire process handles one connection at a time, with a pool of processes spawned in advance — straightforward and very stable, but memory-heavy under high concurrency since every process carries its own full memory footprint. **Worker** is a hybrid: multiple processes, each running multiple threads, so a single process can handle several connections concurrently, using less memory overall than Prefork. **Event**, the modern default, refines Worker further by handling idle keep-alive connections without tying up a full worker thread for each one — closing much of the gap with Nginx's own model.

Nginx's Event-Driven Architecture

Nginx was built specifically to solve what's often called the "C10K problem" — handling ten thousand or more simultaneous connections without the per-connection overhead of Apache's original process/thread model. It runs a small, fixed number of worker processes, each running a single-threaded event loop that uses non-blocking I/O to juggle many connections at once without dedicating a thread or process to each one. This is why Nginx's own memory and CPU usage tends to stay flatter as connection counts climb, compared to a process- or thread-per-connection model.

IIS's Application Pool Model

IIS isolates applications into **Application Pools**, each backed by its own worker process (`w3wp.exe`), so one misbehaving application can't easily crash or starve another running on the same server. Each pool's worker process integrates directly with the .NET runtime's own threading model for handling concurrent requests, and IIS can automatically **recycle** a pool's worker process on a schedule or after memory thresholds, trading a small amount of overhead for resilience against slow memory leaks in long-running applications.

	APACHE (EVENT MPM)	NGINX	IIS
Concurrency model	Processes + threads, with idle connections offloaded	Event-driven, non-blocking, few worker processes	Worker process per application pool, .NET threading inside
Memory under high load	Moderate — improved a lot over Prefork	Low and flat — Nginx's own signature strength	Depends on the application; pools isolate the blast radius
Isolation	Per-process, module-dependent	Per-worker, not per-application by default	Strong — one pool crashing doesn't take down another
Best suited for	Flexible, module-heavy setups	Very high concurrency, static + reverse-proxy workloads	Windows/.NET application hosting

WHY THIS MATTERS BEYOND JUST "WHICH IS FASTER"

These architecture choices directly affect how each server behaves under real, concurrent traffic — not just raw single-request speed. Per Chapter 1's own static/dynamic distinction, a server juggling thousands of connections that are mostly waiting on a slow backend behaves very differently depending on whether idle connections cost a full thread (Apache's older MPMs) or almost nothing (Nginx, Apache's own Event MPM).

"APACHE IS JUST OUTDATED" IS AN OVERSIMPLIFICATION

It's tempting to treat this chapter's comparison as "Nginx's architecture is simply better." In practice, Apache's modern Event MPM closes most of the concurrency gap that made Nginx famous in the first place. The real, lasting differentiator between these servers isn't raw concurrency architecture anymore — it's the configuration philosophy each one is built around, which Chapter 3 covers directly, and that difference is a genuine trade-off, not a strict improvement in either direction.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why Apache's Prefork MPM uses more memory under high concurrency than Nginx's event-driven model, even when both are ultimately serving the same static file to many clients at once.

View solution

EXERCISE 2

A Windows server hosts two separate ASP.NET applications for two different clients. One client's application has a memory leak. Explain, using IIS's Application Pool model, why the other client's application is likely unaffected — and what recycling would do about the leak.

View solution

EXERCISE 3

A colleague claims "Apache is obsolete now that Nginx exists." Using this chapter's own warning box, write a short, accurate response explaining what's true and what's an oversimplification in that claim.

View solution

CHAPTER 2 QUICK REFERENCE

- **Apache** — pluggable MPMs: Prefork (process-per-connection), Worker (processes + threads), Event (modern default, offloads idle connections)
- **Nginx** — a small number of worker processes, each an event-driven, non-blocking loop handling many connections at once
- **IIS** — one worker process (`w3wp.exe`) per Application Pool, isolated from other pools, with scheduled recycling
- Apache's modern Event MPM has closed most of the old Apache-vs-Nginx concurrency gap — the real differentiator now is configuration philosophy (Chapter 3)

Chapter 2 closed on the idea that concurrency architecture no longer meaningfully separates Apache from Nginx — the real, lasting differentiator is how each server expects to be configured. This chapter covers that directly: Apache's per-directory override model, Nginx's deliberately centralized approach, and IIS's GUI/XML system, which turns out to sit closer to Apache's own philosophy than Nginx's.

Apache's Directory-Context Config & .htaccess

Apache's main configuration lives in a central file (`httpd.conf` or `apache2.conf`, depending on distribution), but it also supports **.htaccess** files — small config files placed directly inside a served directory, whose settings apply to that directory (and its subdirectories) without touching the main config or restarting the server. This is genuinely useful in shared-hosting scenarios, where individual users need to override behavior for their own directory without root access to the main config. The cost: on every single request, Apache must check every parent directory of the requested path for a `.htaccess` file, unless this checking is explicitly disabled — a real, measurable per-request filesystem cost in exchange for that flexibility.

Nginx's Centralized, No-Runtime-Override Config

Nginx deliberately has no equivalent to `.htaccess`. All configuration lives in `nginx.conf` (commonly split across multiple files pulled in via `include` directives), and any change requires editing that central configuration and reloading the server. There is no per-directory file Nginx checks at request time — which is exactly the point: this avoids the filesystem-scanning overhead Apache's override mechanism incurs on every request, in exchange for requiring deploy-level access to make any configuration change at all, rather than a lighter-weight per-directory override.

IIS's GUI/XML Config

IIS is configured primarily through the **IIS Manager** GUI, which is itself a front end over XML configuration files — a machine-wide `applicationHost.config`, plus, notably, its own per-directory override mechanism: **web.config** files, which can live inside an individual application's own directory and override settings without touching the machine-wide config. Structurally, this makes IIS closer in spirit to Apache's directory-context model than to Nginx's centralized one, even though IIS's own tooling (a GUI, XML rather than plain-text directives) looks nothing like either of the other two on the surface.

	APACHE	NGINX	IIS
Config format	Plain-text directives	Plain-text directives	XML, via a GUI
Per-directory override	Yes — .htaccess	No — centralized only	Yes — web.config
Change takes effect	Immediately (per-request check)	After a config reload	Immediately for web.config; a full config change may need an app pool recycle
Per-request overhead	Filesystem scan for override files	None — config is fixed at reload time	Filesystem scan for web.config, same trade-off as Apache

CONNECTING THIS TO A REAL DEPLOYMENT

Setting Up a Web Server on Debian's own Apache configuration work was done directly in the main config files — exactly the kind of deploy-level, version-controllable change Nginx requires for every setting. Recognizing which of these two philosophies a given task actually needs — self-service per-directory overrides, or a single, centrally managed, version-controlled config — is the practical question this chapter's comparison is actually building toward.

NO OVERRIDE MECHANISM ISN'T A MISSING FEATURE

It's tempting to read "Nginx has no .htaccess equivalent" as a limitation. It's a deliberate trade-off: Nginx's model assumes whoever manages the server has direct access to the central config and a reload step, which fits a version-controlled, infrastructure-as-code workflow well, but doesn't fit a shared-hosting scenario where individual users need to make changes without deploy access at all. Neither philosophy is strictly better — they're built for different operating assumptions about who's making the change and how.

Hands-On Exercises

EXERCISE 1

A hosting company gives each of its 500 customers their own directory on a shared Apache server, and wants each customer to be able to set their own custom redirect rules without contacting support. Explain why .htaccess is a good architectural fit here, and what the performance trade-off is for making this possible.

View solution

EXERCISE 2

A team runs Nginx and manages its configuration entirely through version-controlled files deployed via an automation tool. Explain why Nginx's lack of a .htaccess-style override mechanism is actually a good fit for this team's workflow, rather than a missing feature.

View solution

EXERCISE 3

Explain why IIS's web.config mechanism is architecturally closer to Apache's .htaccess model than to Nginx's centralized config, despite IIS looking completely different from Apache on the surface (a GUI and XML vs. plain-text directives).

View solution

CHAPTER 3 QUICK REFERENCE

- **Apache** — central config plus optional per-directory `.htaccess` overrides, at the cost of a per-request filesystem scan
- **Nginx** — one central config only, no per-directory override, no per-request scan overhead, changes require a reload
- **IIS** — GUI/XML tooling on the surface, but `web.config` gives it the same per-directory override capability as Apache's `.htaccess`
- Neither philosophy is "better" — they fit different operating assumptions about who needs to make changes and how

Every chapter so far has quietly assumed one server serves one website. In practice, a single physical (or virtual) server very often needs to serve several completely different sites at once — this chapter covers the mechanism each of the three servers uses to do that, under three different names for the same underlying idea.

The Problem: One Server, Many Websites

A server typically has one IP address, listening on port 80/443 — but DNS can point any number of different domain names at that same IP. When a request arrives, every browser sends a `Host` header naming which domain it actually wants, and the web server uses that header to decide which site's content to serve. DNS only gets the request to the right server; splitting it apart by domain from there on is the web server's own job.

Apache's VirtualHost Blocks

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

Apache defines a separate `<VirtualHost>` block per site, each naming its own `ServerName` (and optional `ServerAlias` entries for additional domains that should match the same block) and its own `DocumentRoot`. When a request arrives, Apache matches the incoming `Host` header against each block's `ServerName`/`ServerAlias` to decide which one handles it.

Nginx's Server Blocks

```
server {
    listen 80;
    server_name example.com www.example.com;
    root /var/www/example;
}
```

Nginx's `server {}` block is functionally the same idea as Apache's `<VirtualHost>`, just under different terminology: a `listen` directive for the port, a `server_name` directive matched against the `Host` header, and a `root` for that site's own files. The underlying concept is identical to Apache's — a per-site config block, matched by domain name — even though Nginx's own syntax looks nothing like Apache's directive style.

IIS's Sites

In IIS Manager, each site is configured with one or more **bindings** — a combination of IP address, port, and host name — and every site is tied to a specific Application Pool (Chapter 2). IIS matches an incoming request's `Host` header against each site's bindings the same way Apache and Nginx match against `ServerName` / `server_name`, just configured through the GUI (or the equivalent XML) rather than a text block.

	APACHE	NGINX	IIS
Term used	Virtual Host	Server Block	Site
Matching mechanism	`ServerName` / `ServerAlias` vs. Host header	`server_name` vs. Host header	Bindings (IP/port/hostname) vs. Host header
Config location	A `<VirtualHost>` block, often its own file under `sites-available`	A `server {}` block, often its own file under `sites-enabled`	IIS Manager GUI / `applicationHost.config`
Tied to a process/pool	Shares the same Apache process pool	Shares the same Nginx worker processes	Each Site is tied to its own Application Pool

CONNECTING THIS TO EARLIER CHAPTERS

Setting Up a Web Server on Debian's own real Apache configuration set up exactly one `<VirtualHost>` block for one domain — the same mechanism this chapter describes, just not yet exercised with a second site alongside it. This also sets up Chapter 7 directly: each of these per-site blocks commonly needs its own TLS certificate, since HTTPS/TLS Fundamentals' own certificate material is typically issued per domain, not per server.

DNS DOESN'T SPLIT THE TRAFFIC — THE WEB SERVER DOES

It's a common point of confusion to assume that pointing two different domains at the same server's IP address somehow automatically "just works," with DNS itself routing each domain to different content. DNS only resolves a domain name to an IP address — it has no idea what's running on that server or how many sites it hosts. The actual split, based on the `Host` header included in every request, happens entirely inside the web server itself, via whichever mechanism this chapter just covered.

Hands-On Exercises

EXERCISE 1

Write an Apache VirtualHost block that serves site-a.example from /var/www/site-a and a separate one that serves site-b.example from /var/www/site-b, both on port 80.

View solution

EXERCISE 2

A user points two different domains at the same server's IP address via DNS, then is confused when both domains show identical content instead of two different sites. Using this chapter's own warning box, explain what's actually missing.

View solution

EXERCISE 3

Explain, in your own words, why an IIS Site being tied to its own Application Pool is a meaningfully different guarantee than an Apache VirtualHost or Nginx server block sharing the same pool of worker processes as every other site on that server.

View solution

CHAPTER 4 QUICK REFERENCE

- The same idea, three names: Apache's **VirtualHost**, Nginx's **server block**, IIS's **Site**
- All three match an incoming request's **Host header** against a configured name/binding to decide which site handles it
- DNS only resolves a name to an IP — the web server itself performs the actual per-domain split
- IIS Sites are each tied to their own Application Pool (Chapter 2); Apache/Nginx sites on the same server typically share the same process pool

Chapter 1 described serving a static file as simply "read it from disk, send it back" — true as far as it goes, but there's more going on underneath. This chapter covers three details every one of the three servers has to handle for static content: telling the browser what kind of file it just received, deciding what to serve when a URL names a folder rather than a file, and what happens when there's nothing obvious to serve at all.

MIME Types: Telling the Browser What It's Looking At

Every HTTP response includes a `Content-Type` header — the MIME type — telling the browser how to interpret the bytes that follow (as HTML, as CSS, as a JPEG image, and so on). Each server maps file extensions to MIME types using its own lookup table: Apache and Nginx both ship a `mime.types` file mapping common extensions to their MIME type, extendable via Apache's `AddType` directive or Nginx's own `types {}` block; IIS maintains its own MIME type mappings through IIS Manager or the equivalent XML config. An extension with no matching entry typically falls back to a generic type like `application/octet-stream`, which prompts most browsers to offer the file as a download rather than trying to render it — a frequent, easy-to-miss cause of "why is my browser downloading this instead of showing it."

Directory Indexes: What Happens When a URL Names a Folder

Requesting `/blog/` rather than a specific file requires the server to decide what to actually serve. All three maintain an ordered list of candidate filenames to check inside that directory: Apache's `DirectoryIndex` directive (commonly `index.html index.php`, checked in order), Nginx's `index` directive (the same idea, its own directive name), and IIS's **Default Document** feature, configured via IIS Manager, listing candidates like `Default.htm` or `index.html` in priority order. All three are solving the exact same problem — what to serve for a bare directory request — just under a different directive name each.

Directory Listings: What Happens With No Matching Index File

If none of the configured index candidates exist in a requested directory, each server has a separate, distinct setting controlling what happens next: generate an automatic listing of the directory's actual contents, or return an error. Apache's `mod_autoindex` can generate this listing if explicitly enabled; Nginx's own `autoindex` directive is `off` by default; IIS's **Directory Browsing** feature is likewise off by default. Enabling this is occasionally useful (a simple internal file share), but it's also a well-known way to accidentally expose a directory's full contents to anyone who requests it.

	APACHE	NGINX	IIS
MIME type mapping	`mime.types` + `AddType`	`mime.types` + `types {}`	MIME Types config (GUI/XML)
Default document directive	`DirectoryIndex`	`index`	Default Document (GUI)
Directory listing default	Off (module not typically enabled)	Off (`autoindex off`)	Off (Directory Browsing disabled)
Unknown extension fallback	`application/octet-stream`	`application/octet-stream`	`application/octet-stream`

A REAL DEBUGGING SCENARIO THIS EXPLAINS

A stylesheet that loads successfully (no 404 in the browser's network tab) but visibly has no effect on the page is very often a MIME type problem, not a missing-file problem — if the server sends it back as `text/plain` or `application/octet-stream` instead of `text/css`, many browsers refuse to apply it as a stylesheet at all. Checking the actual `Content-Type` header, not just whether the file loaded, is the right first step whenever this happens.

DIRECTORY LISTING IS A REAL, COMMON MISCONFIGURATION, NOT A HARMLESS CONVENIENCE

All three servers default to directory listing off precisely because leaving it enabled — even unintentionally, by installing a module or checking a box without realizing what it does — can expose a directory's exact file names to anyone who requests that folder's URL, including backup files, configuration snippets, or other content nobody meant to publish. This is a genuine, well-documented misconfiguration category, not a stylistic preference — Chapter 8's own hardening coverage returns to this directly.

Hands-On Exercises

EXERCISE 1

A server serves a .woff2 font file with no matching MIME type entry configured, so it falls back to application/octet-stream. Explain what a browser is likely to do when it requests this file as part of loading a webpage's CSS, and why.

View solution

EXERCISE 2

A directory contains index.php but not index.html, and the server's directory-index list is configured as "index.html index.php" in that order. Explain what actually gets served when a bare directory URL is requested, and why the order of the list matters.

View solution

EXERCISE 3

A developer enables directory listing on a production server "temporarily, just to quickly check what files are in a folder from the browser," then forgets to disable it. Explain, using this chapter's own warning box, what could actually go wrong if this is left enabled.

View solution

CHAPTER 5 QUICK REFERENCE

- **MIME types** tell the browser how to interpret a response's body — an unmapped extension falls back to `application/octet-stream`, often triggering a download instead of rendering
- **Directory index candidates** — Apache's `DirectoryIndex`, Nginx's `index`, IIS's Default Document — all check an ordered list of filenames for a bare directory request
- **Directory listing** is off by default on all three servers, for good reason — it's a real, common way to accidentally expose a folder's contents

Chapter 1 described dynamic content as something a web server "hands off" to a separate application process. This chapter covers the actual mechanism behind that handoff — reverse proxying — kept deliberately foundational here, since Nginx In Depth is where this specific territory gets its own full treatment.

What a Reverse Proxy Actually Does

A reverse proxy is a server that sits in front of one or more backend servers, receives client requests on their behalf, forwards each request to a backend, and relays the backend's response back to the client — who never directly talks to the backend at all, and typically has no idea it exists. This is the opposite direction from a **forward proxy**, which represents a client (hiding the client from whatever server it's talking to) rather than representing a backend server to the outside world.

Apache as a Reverse Proxy

```
ProxyPass /app http://127.0.0.1:5000/
ProxyPassReverse /app http://127.0.0.1:5000/
```

Apache's `mod_proxy` module (with `mod_proxy_http` for HTTP backends) enables the `ProxyPass` / `ProxyPassReverse` directive pair: `ProxyPass` forwards matching requests to the backend, and `ProxyPassReverse` rewrites any redirect the backend sends back so it still points at the proxy's own address rather than leaking the backend's internal one.

Nginx as a Reverse Proxy

```
location /app/ {
    proxy_pass http://127.0.0.1:5000/;
}
```

Nginx's own `proxy_pass` directive is genuinely one of Nginx's single most common real-world uses — serving as the front door to a Node.js, Python, or Java application process is a large part of why Nginx shows up so often in modern deployments. This is exactly the territory Nginx In Depth exists to cover thoroughly.

IIS as a Reverse Proxy

IIS does not include reverse-proxy capability by default — it requires installing a separate module, **Application Request Routing (ARR)**, alongside the **URL Rewrite** module it depends on. Once installed, ARR can forward requests to one or more backend servers, similar in spirit to Apache's or Nginx's own directives, but as an add-on rather than a built-in core capability — a genuine architectural difference worth noting, not just a naming difference.

Load Balancing Basics

Once more than one backend instance exists, a reverse proxy can also **load balance** — distributing requests across several backend instances rather than sending every request to just one. **Round robin** (cycling through each backend in turn) is the simplest algorithm and the default for all three: Apache's `mod_proxy_balancer`, Nginx's `upstream {}` block naming multiple backend addresses, and IIS ARR's own server-farm feature, which groups multiple backend servers together behind one routing rule.

	APACHE	NGINX	IIS
Reverse proxy mechanism	`mod_proxy` + `ProxyPass`	`proxy_pass`	Application Request Routing (ARR)
Built in or add-on	Module, commonly enabled	Built in, core use case	Separate module install required
Load balancing	`mod_proxy_balancer`	`upstream {}` block	ARR server farms
Default algorithm	Round robin	Round robin	Round robin (weighted options available)

WHERE THIS SHOWS UP BEYOND A SINGLE SERVER

Kubernetes' own Ingress controllers are very commonly built on top of exactly this mechanism — an Nginx (or similar) reverse proxy configured to route incoming cluster traffic to the right backend Pod, the same `proxy_pass`-style forwarding this chapter just introduced, just automated and driven by Kubernetes' own configuration instead of a hand-edited config file.

A REVERSE PROXY ISN'T ONLY FOR LOAD BALANCING MULTIPLE SERVERS

It's easy to assume reverse proxying only matters once there are several backend instances to distribute traffic across. In practice, a reverse proxy in front of a single backend is extremely common too — often purely to gain TLS termination (Chapter 7), response caching, or protocol translation (e.g. exposing a plain-HTTP Node.js app to the outside world over HTTPS, with the proxy handling the encryption layer the backend never has to know about). Load balancing is one reason to reverse-proxy, not the only one.

Hands-On Exercises

EXERCISE 1

Write the Nginx configuration (a location block) that reverse-proxies requests under /api/ to a backend application running on 127.0.0.1:3000.

View solution

EXERCISE 2

A team runs a single Node.js application with no plans to run multiple instances of it. A teammate asks why they'd bother putting Nginx in front of it as a reverse proxy if there's nothing to load balance. Give an accurate answer using this chapter's own material.

View solution

EXERCISE 3

Explain why IIS needing a separate module (ARR) for reverse proxying, while Apache and Nginx both support it as part of their normal module ecosystem or core functionality, is a genuine architectural difference rather than just a difference in terminology.

View solution

CHAPTER 6 QUICK REFERENCE

- A **reverse proxy** forwards client requests to a backend and relays the response back — the client never talks to the backend directly
- Apache: `ProxyPass` / `ProxyPassReverse`. Nginx: `proxy_pass` — its own single most common real-world use case. IIS: Application Request Routing (ARR), a separate module install
- **Load balancing** (round robin, by default on all three) distributes requests across multiple backend instances once more than one exists
- Reverse proxying is just as often used for a **single** backend — TLS termination, caching, protocol translation — not only for load balancing several
- This chapter is deliberately foundational — **Nginx In Depth** covers this territory in full

Chapter 6 named TLS termination as one of the standalone reasons to put a reverse proxy in front of a single backend. This chapter is that idea, addressed directly — not a re-explanation of what a certificate or a TLS handshake actually is (HTTPS/TLS Fundamentals already covers that thoroughly), but specifically where and how each of the three web servers plugs into that handshake.

What "Termination" Actually Means

TLS termination is the point in the request path where encrypted traffic gets decrypted back into plain HTTP — the server or reverse proxy handling that decryption is said to "terminate" the TLS connection there. This is distinct from **TLS passthrough**, where a proxy forwards the still-encrypted traffic onward without decrypting it itself, leaving termination to happen further down the chain, at the actual backend. Most of the setups this course discusses terminate TLS at the web server or reverse-proxy layer, with the connection to any backend behind it handled separately.

Apache's TLS Configuration

```
<VirtualHost *:443>
ServerName example.com
    SSLEngine on
    SSLCertificateFile /etc/ssl/certs/example.com.crt
    SSLCertificateKeyFile /etc/ssl/private/example.com.key
</VirtualHost>
```

Apache's `mod_ssl` module adds the `SSLCertificateFile`/`SSLCertificateKeyFile` directives (and `SSLCertificateChainFile` where an intermediate chain is needed) inside a `<VirtualHost>` listening on port 443. Per Chapter 4's own per-VirtualHost model, a server hosting several domains typically needs its own certificate configuration per VirtualHost, unless **SNI** (Server Name Indication) is relied on to let one shared connection still resolve to the right certificate per domain — supported by all three servers covered in this course.

Nginx's TLS Configuration

```
server {
    listen 443 ssl;
    server_name example.com;
    ssl_certificate /etc/ssl/certs/example.com.crt;
    ssl_certificate_key /etc/ssl/private/example.com.key;
}
```

Nginx's equivalent lives inside a `server {}` block: `listen 443 ssl` to accept HTTPS connections, plus `ssl_certificate` / `ssl_certificate_key` pointing at the certificate and private key files — the same underlying concept as Apache's directives, again per-server-block, matching Chapter 4's own per-site model.

IIS's TLS Configuration

IIS handles certificates differently at a structural level: rather than pointing at certificate files on disk, IIS integrates with the **Windows Certificate Store** — certificates are imported into that store, then an **https binding** (Chapter 4's own binding concept) is configured in IIS Manager to pair a specific certificate from the store with a specific site. This is a genuine architectural difference from Apache's and Nginx's file-based approach, not just a different way of pointing at the same kind of file.

	APACHE	NGINX
Certificate source	Files on disk	Files on disk
Config mechanism	`SSLCertificateFile` / `SSLCertificateKeyFile`	`ssl_certificate` / `ssl_certificate_`
Scope	Per VirtualHost	Per server block
SNI support	Yes	Yes

> **THIS CHAPTER ASSUMES HTTPS/TLS FUNDAMENTALS**
>
> What a certificate authority actually verifies, what the TLS handshake itself involves, and why a certificate chain matters are all covered in HTTPS/TLS Fundamentals — this chapter deliberately doesn't repeat that material, and instead only covers where each of these three specific web servers plugs a certificate into that already-understood process.

TERMINATION DOESN'T AUTOMATICALLY MEAN THE WHOLE PATH STAYS ENCRYPTED

It's easy to assume that once HTTPS is set up at the web server, "the connection is encrypted" end to end. In practice, once TLS is terminated at the web server or reverse-proxy layer, the connection onward to an actual backend application — per Chapter 6's own reverse-proxy material — is very often plain, unencrypted HTTP internally, unless deliberately re-encrypted for that hop. This is a common and reasonable choice inside a trusted private network, but it's a real design decision worth being aware of, not an automatic guarantee that comes bundled with "HTTPS is enabled."

Hands-On Exercises

EXERCISE 1

Write the Nginx server block that terminates HTTPS for shop.example, using certificate files at /etc/ssl/certs/shop.crt and /etc/ssl/private/shop.key.

View solution

EXERCISE 2

A colleague assumes that because their site shows a padlock and "https://" in the browser, every hop of that request's journey — including from the reverse proxy to the backend application — must be encrypted. Using this chapter's own warning box, explain what's actually true here.

View solution

EXERCISE 3

Explain why IIS's Windows Certificate Store approach is a genuine architectural difference from Apache's and Nginx's file-based certificate configuration, rather than just a different syntax for pointing at the same underlying thing.

View solution

CHAPTER 7 QUICK REFERENCE

- **TLS termination** — the point where encrypted traffic is decrypted back to plain HTTP, distinct from passthrough
- Apache: `SSLCertificateFile` / `SSLCertificateKeyFile` per VirtualHost. Nginx: `ssl_certificate` / `ssl_certificate_key` per server block. IIS: an HTTPS binding referencing the Windows Certificate Store
- All three support **SNI**, letting one shared connection resolve to the right certificate per domain
- Terminating TLS at the web server does **not** automatically mean the hop to a backend application stays encrypted — that's a separate, deliberate decision

Chapter 5 flagged directory listing as a real, common misconfiguration. This chapter widens that lens: what each server records about every request by default, how to restrict access at the web-server layer itself before a request ever reaches the application, and a handful of other low-cost hardening steps worth knowing about.

Access Logs & Error Logs

Every request handled is normally recorded in an **access log** — client IP, timestamp, the request line itself, the response status code, and typically the response size — while server-side problems (a misconfiguration, a backend failing to respond) go to a separate **error log**. Apache uses `access.log` / `error.log`, with the exact fields recorded controlled by a configurable `LogFormat`. Nginx uses its own `access_log` / `error_log` directives, similarly format-configurable. IIS records requests in the W3C Extended Log File Format by default, configured through IIS Manager. All three capture broadly the same core information, just formatted and configured differently.

Basic Access Control

Restricting who can reach a given path can happen at the web-server layer itself, before a request ever reaches the application behind it. Apache's `Require` directive (via `mod_authz_host`) can restrict access by IP, and `.htpasswd`-backed Basic Authentication can require a username/password. Nginx uses its own `allow`/`deny` directives for IP restriction and `auth_basic` for password protection. IIS offers the equivalent through its **IP Address and Domain Restrictions** feature, plus Basic or Windows Authentication configured per site. In every case, a request that fails this check never reaches the application at all — the rejection happens at the web server itself.

Basic Hardening: Hiding Version Information

```
# Apache (httpd.conf / apache2.conf)
ServerTokens Prod
ServerSignature Off

# Nginx (nginx.conf)
server_tokens off;
```

By default, all three servers reveal their own name and version number in response headers or error pages — Apache's `ServerTokens` / `ServerSignature` and Nginx's `server_tokens off` suppress this; IIS requires an add-on (commonly via URL Rewrite rules) to strip its own version string similarly. The goal is reducing the information an attacker can gather during initial reconnaissance — knowing the exact server version can point directly at known, unpatched vulnerabilities for that specific version.

Revisiting Directory Listing

Chapter 5's own directory-listing warning belongs on this same checklist: confirming it's genuinely disabled (not just left at whatever the default happens to be) is one more low-cost step in the same category as version hiding and access control — each one closes off a small piece of information or access an attacker could otherwise use.

	APACHE	NGINX	IIS
Access/error logs	`access.log` / `error.log`, configurable `LogFormat`	`access_log` / `error_log` directives	W3C Extended Log File Format
IP-based access control	`Require` (mod_authz_host)	`allow` / `deny`	IP Address and Domain Restrictions
Password auth	`.htpasswd` + Basic Auth	`auth_basic`	Basic/Windows Authentication
Hide version info	`ServerTokens` / `ServerSignature`	`server_tokens off`	Requires an add-on (e.g. URL Rewrite)

WHERE THESE LOGS GO NEXT

Raw access and error logs are exactly the kind of data an observability stack (Observability's own material) is built to collect, aggregate, and alert on — this chapter covers what each server records natively; what happens to that data afterward is a separate, later concern. Similarly, hiding version information and tightening access control are the same category of step covered from the attacker's own perspective in Penetration Testing Methodology's reconnaissance material.

HIDING VERSION INFORMATION IS NOT, BY ITSELF, SECURITY

Suppressing a server's version string is a real, worthwhile, low-cost step — but it's **security through obscurity**: it slows down casual or fully automated scanning, not a determined attacker willing to fingerprint the server through other means. It is never a substitute for actually keeping the server patched and correctly configured — the same caution OWASP Top 10's own material applies broadly to any control whose entire value is "the attacker doesn't immediately know something," rather than a control that actually blocks an attack outright.

Hands-On Exercises

EXERCISE 1

Write the Nginx configuration that restricts access to a /admin/ path to only the IP address 203.0.113.10, denying everyone else.

View solution

EXERCISE 2

A team disables ServerTokens/server_tokens on their production servers and considers the server "properly secured" as a result, deprioritizing an overdue security patch. Using this chapter's own warning box, explain what's wrong with this reasoning.

View solution

EXERCISE 3

Explain, in your own words, why restricting access to a sensitive path at the web-server layer (via IP restriction or Basic Auth) is meaningfully different from relying on the application itself to check permissions before showing that content.

View solution

CHAPTER 8 QUICK REFERENCE

- All three servers log requests by default — Apache/Nginx via configurable plain-text formats, IIS via W3C Extended Log File Format
- **Access control at the web-server layer** rejects disallowed requests before they ever reach the application
- **Hiding version info** (ServerTokens/server_tokens off) reduces reconnaissance value but is security through obscurity, not a real fix
- Directory listing (Chapter 5) belongs on this same hardening checklist — confirm it's genuinely disabled, not just defaulted

Chapter 2 compared how each server handles many connections at once, and Chapter 6 mentioned caching in passing as one reason to reverse-proxy a single backend. This chapter covers three concrete performance techniques any of the three servers can apply, largely independent of the architecture debates earlier chapters focused on — and where each one can actually backfire if applied without thinking about what it's doing.

Keep-Alive Connections

Opening a new TCP connection (and, over HTTPS, repeating the TLS handshake from Chapter 7) for every single request is expensive. **Keep-alive** reuses one already-open connection for multiple requests instead, avoiding that repeated setup cost. Apache controls this with `KeepAlive`, `KeepAliveTimeout`, and `MaxKeepAliveRequests`; Nginx with `keepalive_timeout` and `keepalive_requests`; IIS through its own connection timeout settings in IIS Manager. All three enable keep-alive by default today — the main tuning question is how long to hold an idle connection open before closing it, trading a small amount of held-open resources against avoiding a future reconnection cost.

Compression

```
# Nginx
gzip on;
gzip_types text/css application/javascript text/html;
```

Compressing a response body (typically with gzip, or the newer and generally more efficient brotli) before sending it trades a small amount of server CPU time for meaningfully fewer bytes sent over the network — usually a clear win for text-based content like HTML, CSS, and JavaScript. Apache enables this via `mod_deflate`; Nginx via its `gzip` directive plus a `gzip_types` list naming which content types to compress; IIS through its Dynamic Content Compression and Static Content Compression settings in IIS Manager.

Caching

Caching happens at two different levels. **Browser caching** uses response headers like `Cache-Control` and `Expires` to tell the browser it can reuse a previously downloaded asset without re-requesting it at all for some period of time — set directly by the web server for static files. **Server-side or proxy caching** — Chapter 6's own caching mention, revisited here — stores a copy of a response on the server or reverse proxy itself, so a subsequent identical request can be answered from that stored copy without hitting the backend application again at all. Apache offers this via `mod_cache`; Nginx via `proxy_cache`; IIS via its own Output Caching feature.

	APACHE	NGINX	IIS
Keep-alive	`KeepAlive` / `KeepAliveTimeout`	`keepalive_timeout` / `keepalive_requests`	Cor tim sett Mai
Compression	`mod_deflate`	`gzip` / `gzip_types`	Dyr Cor Cor
Server-side caching	`mod_cache`	`proxy_cache`	Out Cac

THE SERVER-SIDE COUNTERPART TO A BROWSER-METRICS COURSE

Performance & Core Web Vitals covers page-load performance from the browser's own side — what a user actually experiences once a response arrives. This chapter's material is the server-side counterpart: keep-alive, compression, and caching all directly shrink how long it takes for that response to arrive in the first place, before any of Core Web Vitals' own browser-side metrics even start measuring.

"TURN EVERYTHING ON" IS NOT A SAFE DEFAULT

Blanket-enabling every technique in this chapter without thinking about what it's actually doing can backfire in two distinct ways. Compressing already-compressed content — images, video, anything already in a binary compressed format — wastes CPU for little or no size reduction, since there's rarely much redundancy left to squeeze out. Far more seriously, caching dynamic or personalized content incorrectly — serving one user's cached, logged-in page to a different user who requests the same URL — is a real, well-documented class of caching bug that can leak private data between users, not merely a performance nitpick. Caching rules need to account for exactly what varies between requests, not just whether a URL matches.

Hands-On Exercises

EXERCISE 1

Write the Nginx directives that enable gzip compression for text/css, application/javascript, and text/html content.

View solution

EXERCISE 2

A team enables gzip compression for every content type on their server, including .jpg and .mp4 files. Explain, using this chapter's own warning box, why this is unlikely to help and may actually make things slightly worse.

View solution

EXERCISE 3

A site enables server-side/proxy caching for a page that shows each logged-in user their own account dashboard, using the URL alone as the cache key. Explain what could go wrong, and why this is a more serious problem than a typical performance misconfiguration.

View solution

CHAPTER 9 QUICK REFERENCE

- **Keep-alive** — reuse one TCP/TLS connection for multiple requests, avoiding repeated handshake overhead
- **Compression** (gzip/brotli) — trades server CPU for fewer bytes over the wire; a clear win for text, not for already-compressed binary content
- **Browser caching** (Cache-Control/Expires) vs. **server-side/proxy caching** (Chapter 6 revisited) — two different levels solving two different problems
- Caching dynamic/personalized content incorrectly can leak one user's data to another — a real security bug, not just a performance issue

Every chapter so far compared Apache, Nginx, and IIS along one specific axis — architecture, configuration philosophy, virtual hosting, static content, reverse proxying, TLS, hardening, performance. This capstone pulls all eight together into an actual decision framework, then applies it to three realistic scenarios, closing the course.

The Real Question This Course Has Been Building Toward

Chapter 2's own warning box already rejected "which server is objectively best" as the wrong question. The real question, running underneath every chapter since, has been: **which server's own philosophy actually fits this specific job's real constraints?** A framework for answering that consistently is what this capstone provides.

A Decision Framework

- **Platform constraint** — is this a Windows/.NET environment already? (Chapters 2 & 7's own IIS material)
- **Configuration model** — does this job need self-service, per-directory overrides, or a single, centrally managed, version-controlled config? (Chapter 3)
- **Concurrency profile** — how much simultaneous traffic, and how much of it is short-lived static requests vs. long-held dynamic connections? (Chapter 2)
- **Reverse-proxy/load-balancing needs** — is this server the front door to one backend, several, or none at all? (Chapter 6)
- **Certificate management model** — file-based, or integrated with an existing certificate store? (Chapter 7)

Scenario 1 — An Internal .NET Line-of-Business App

A company builds an internal tool for 50 employees, written in ASP.NET, running entirely inside a Windows-based internal network. The platform constraint dominates here: the team is already deep in the Windows/.NET ecosystem, and IIS's own tight integration with that stack (Chapter 2's Application Pool model, Chapter 7's Windows Certificate Store) is a natural fit rather than a fight against the grain. Neither Apache's nor Nginx's own strengths (module ecosystem, high-concurrency performance) are the deciding factor here — the platform is.

Scenario 2 — A High-Traffic Public API

A public-facing API expects heavy, sustained concurrent traffic, runs as several containerized backend instances that need traffic distributed across them, and needs response caching to reduce load on those backends. This profile points squarely at Nginx: Chapter 2's own event-driven concurrency strength, Chapter 6's reverse-proxying and load-balancing (genuinely Nginx's own signature use case), and Chapter 9's `proxy_cache` all line up directly with what this job actually needs — and this is exactly the scenario Nginx In Depth exists to go further into.

Scenario 3 — Shared Hosting for Many Customers

A hosting company serves hundreds of customers on one physical server, each needing to set their own redirect rules and basic config without contacting support. Chapter 3's own `.htaccess` material makes the choice here: Apache's directory-context override model is architecturally built for exactly this self-service requirement, in a way Nginx's centralized-only config deliberately isn't.

SCENARIO	CHOICE	DECIDING CHAPTER(S)
1 — Internal .NET LOB app	IIS	Chapter 2 (Application Pools), Chapter 7 (Certificate Store)
2 — High-traffic public API	Nginx	Chapter 2 (concurrency), Chapter 6 (reverse proxy/load balancing), Chapter 9 (caching)
3 — Shared hosting, many customers	Apache	Chapter 3 (.htaccess self-service config)

WHERE TO GO FROM HERE

Setting Up a Web Server on Debian already walked through a real single-site Apache deployment — a simpler version of Scenario 3's own territory without the multi-tenant self-service angle. Scenario 2's own reverse-proxy and load-balancing needs are exactly the territory Nginx In Depth exists to cover in full detail, well beyond this course's own deliberately foundational Chapter 6.

THIS FRAMEWORK DESCRIBES FIT, NOT A PERMANENT RANKING

None of the three scenarios above crowned one server "the winner" — each one fit a specific set of real constraints that could easily point somewhere else if those constraints changed. A team's own platform commitments, traffic patterns, and operational habits shift over time, and the right answer for a given job can shift right along with them — the value of this chapter's framework is asking the right questions freshly each time, not memorizing a fixed answer.

Hands-On Exercises

EXERCISE 1

A small nonprofit runs one static informational website (no dynamic content at all) on a single low-traffic server, with one volunteer maintaining it who is comfortable editing a config file directly. Using this chapter's own decision framework, which server would you recommend, and why?

View solution

EXERCISE 2

A company currently uses IIS for an internal .NET tool (Scenario 1), but two years later rewrites that same tool as a cross-platform, containerized service intended for public internet traffic. Explain, using this chapter's own framework, whether the original IIS recommendation should still hold, and why.

View solution

EXERCISE 3

Write a short chapter-attribution summary (2-3 sentences) explaining how Chapters 2, 3, 6, and 7 each contributed a distinct question to this capstone's own decision framework.

View solution

COURSE COMPLETE

Web Servers Fundamentals — 10 of 10 chapters complete. Nginx In Depth remains outstanding as this subject's own next planned course.

CHAPTER 10 QUICK REFERENCE

- The decision framework: **platform constraint, configuration model, concurrency profile, reverse-proxy/load-balancing needs, certificate management model**
- **IIS** fits an existing Windows/.NET platform commitment; **Nginx** fits high-concurrency, reverse-proxy-heavy, cache-reliant workloads; **Apache** fits self-service, per-directory-override needs like shared hosting
- This framework describes **fit for a specific job's real constraints**, not a fixed ranking — the right answer can change as those constraints change