WEB SERVERS

IIS In Depth

Application Pools & Recycling · The web.config/applicationHost.config Model

The Integrated Pipeline · Sites, Virtual Directories & Applications

URL Rewrite · TLS/HTTPS & SNI

Windows Authentication & Hardening · Performance & Failed Request Tracing

Capstone: A Production IIS Deployment

Philip Osztromok

Generated with Claude

Table of Contents

IIS In Depth — 10 Chapters

Web Servers Fundamentals compared Apache, Nginx, and IIS across architecture, configuration philosophy, virtual hosting, TLS, and performance — but deliberately kept its own IIS material introductory in several specific places. This course exists to go deep on exactly those places, with one deliberate exception named honestly up front, rather than forced into matching its siblings' shape.

What Web Servers Fundamentals Already Covered

IIS shows up in exactly as many of that course's chapters as Apache did — seven — genuinely more than Nginx's three, and for a similar underlying reason: like Apache's `.htaccess`, IIS has its own real per-directory override mechanism (`web.config`), which Web Servers Fundamentals Chapter 3 found structurally closer to Apache's own philosophy than to Nginx's centralized one. Chapter 2 introduced Application Pools and their `w3wp.exe` worker process conceptually, including scheduled/threshold-based recycling, without configuring either in depth. Chapter 3 introduced the GUI/XML configuration model — `applicationHost.config` machine-wide, `web.config` per-directory — without covering real inheritance behavior. Chapter 4 covered Sites and bindings at a basic level. Chapter 6 named **Application Request Routing (ARR)** as IIS's own reverse-proxy capability — genuinely an add-on module, not built in the way Apache's `mod_proxy` or Nginx's `proxy_pass` are. Chapter 7 covered the Windows Certificate Store and HTTPS bindings generically. Chapter 8 named IP/Domain Restrictions and Basic/Windows Authentication without real directive depth. Chapter 9 named Dynamic/Static Content Compression and Output Caching, briefly and comparatively.

Why This Course Is IIS-Only

The comparative question — Apache vs. Nginx vs. IIS, and which one actually fits a given job — already belongs to Web Servers Fundamentals, and its own capstone decision framework is the right place to answer it. This course assumes that question is already settled: either IIS is genuinely the right fit for the job in front of you (very often because the application itself is a Windows/.NET one), or you simply want to understand IIS itself in real depth. Either way, every chapter from this point on is IIS configuration and IIS-specific behavior — no more comparing.

A Deliberate Scope Boundary: No Dedicated Reverse-Proxy Chapter

Both Apache In Depth and Nginx In Depth devote real chapters to reverse proxying and load balancing, because `mod_proxy` and `proxy_pass` are core, built-in capabilities of those two servers. IIS's own ARR module is a genuinely optional add-on most IIS installations never install at all — IIS's primary job in the overwhelming majority of real deployments is hosting ASP.NET/.NET applications directly, not proxying to something else. Giving ARR its own full chapter here would misrepresent how central it actually is to IIS, compared to how central reverse proxying genuinely is to Apache and Nginx. This course goes deep on what IIS itself does natively instead.

What This Course Actually Adds

- **Chapter 2** — Application Pool identities, real recycling triggers, and live health monitoring, not just the conceptual overview
- **Chapter 3** — `web.config` / `applicationHost.config` inheritance in real depth
- **Chapter 4** — the Integrated Pipeline, and Managed vs. Native modules — a genuinely new topic Fundamentals never touched at all
- **Chapter 5** — Sites, virtual directories, and application boundaries in depth
- **Chapter 6** — the URL Rewrite module in depth (distinct from ARR — Rewrite alone, without ARR, is genuinely common and doesn't require proxying anything)
- **Chapter 7** — HTTPS bindings, SNI, and the centralized certificate store, well beyond Chapter 7's generic treatment
- **Chapter 8** — Windows Authentication, Request Filtering, and IP restrictions with real directive depth
- **Chapter 9** — performance tuning, logging, and Failed Request Tracing — another genuinely new topic
- **Chapter 10** — a capstone that designs and hardens one complete, production-shaped IIS deployment

TOPIC	WEB SERVERS FUNDAMENTALS	THIS COURSE
Application Pools	Conceptual overview (Ch.2)	Identities, recycling triggers, health monitoring (Ch.2)
Configuration model	GUI/XML introduced (Ch.3)	Real inheritance across `web.config` / `applicationHost.config` (Ch.3)
Module pipeline	Not covered	Integrated Pipeline, Managed vs. Native modules (Ch.4)
Reverse proxy (ARR)	Named as an add-on (Ch.6)	Not covered — a deliberate scope boundary, see above
Comparison to Apache/Nginx	Central focus of the whole course	None — IIS only

WHERE THIS SHOWS UP BEYOND AN ON-PREMISES SERVER

Azure App Service's Windows-based hosting tier runs on IIS under the hood — the Application Pool model, the `web.config`-driven configuration, and the module pipeline this course covers are the same real mechanisms behind a large share of enterprise .NET applications running in Azure today, not just an on-premises Windows Server concern.

"IN DEPTH" DOESN'T MEAN "START FROM ZERO, BUT HARDER"

This course assumes the basic IIS concepts from Web Servers Fundamentals — Sites, bindings, and the Application Pool model — are already familiar. It doesn't re-teach them from scratch; it builds directly on top of them, and leans on raw `web.config`/XML syntax throughout for precision, naming the IIS Manager GUI equivalent where it matters. Anyone who hasn't gone through Web Servers Fundamentals' own IIS material (Chapters 2, 3, 4, 6, 7, 8, and 9) may want to revisit those chapters first.

Hands-On Exercises

EXERCISE 1

In your own words, explain why this course doesn't include an Apache or Nginx comparison anywhere, even though Web Servers Fundamentals covered all three servers side by side.

View solution

EXERCISE 2

This chapter names seven Web Servers Fundamentals chapters that touched IIS directly — the same count as Apache, and more than Nginx's three. Name at least five of them, what each left deliberately shallow, and explain in one sentence why IIS needed about as much of Fundamentals' own material as Apache did.

View solution

EXERCISE 3

Explain, in your own words, why this course has no dedicated reverse-proxy/ARR chapter, unlike its Apache In Depth and Nginx In Depth siblings, using this chapter's own scope reasoning.

View solution

CHAPTER 1 QUICK REFERENCE

- This course is **IIS-only** — the Apache/Nginx comparison already belongs to Web Servers Fundamentals
- Seven Fundamentals chapters touched IIS directly — the same count as Apache, since both have their own per-directory config-override mechanism (`web.config` / `.htaccess`), unlike Nginx
- **No dedicated reverse-proxy chapter** — ARR is a genuine optional add-on for IIS, unlike `mod_proxy` / `proxy_pass`'s built-in role for Apache/Nginx
- Assumes familiarity with basic IIS concepts (Sites, bindings, Application Pools) from Web Servers Fundamentals
- Nine chapters ahead: Application Pools, configuration inheritance, the module pipeline, sites/virtual directories, URL Rewrite, TLS/SNI, authentication, performance/troubleshooting, and a capstone

Web Servers Fundamentals Chapter 2 described Application Pools conceptually: each one backed by its own `w3wp.exe` worker process, integrating with the .NET runtime, recycled "on a schedule or after memory thresholds." This chapter makes that real — the actual security identity a pool runs under, the real configurable recycling triggers, and a genuine protective circuit-breaker that can look exactly like a server outage if you don't know it exists.

Application Pool Identities

```
appcmd set apppool "MyAppPool" /processModel.identityType:ApplicationPoolIdentity
```

Every pool's worker process runs under a security identity, and the choice genuinely matters for isolation between applications on the same server. **ApplicationPoolIdentity** — the modern default — gives each pool its own unique, dynamically-created virtual account with its own unpredictable SID, meaning file-system and resource permissions can be granted to one specific pool's identity without any other pool sharing that access. Older or legacy configurations sometimes use **NetworkService**, **LocalService**, or **LocalSystem** — broader, shared built-in accounts where multiple pools running under the same identity can, in principle, access resources that were only ever meant for one of them. A custom domain or local account is also possible, typically only when a specific external resource (a network share, a database) requires it.

Recycling Triggers In Depth

```
<recycling>
<periodicRestart time="1.05:00:00" privateMemory="1048576" requests="0" />
</recycling>
```

`time` is the regular scheduled-recycle interval — the real IIS default is **29 hours** (`1.05:00:00`, one day and five hours), an oddly specific number chosen deliberately so scheduled recycles don't fall at the same clock time every single day. `privateMemory` (in KB) and a request-count limit are both disabled (`0`) unless explicitly configured, as shown here for a 1GB private-memory ceiling. By default, recycling is **overlapped**: a new worker process starts and begins accepting requests *before* the old one is told to stop, and the old process only exits once it finishes any in-flight requests — a scheduled recycle doesn't have to mean a dropped request or a visible interruption.

Rapid-Fail Protection — A Real Circuit Breaker

```
<failure rapidFailProtection="true" rapidFailProtectionInterval="00:05:00"
rapidFailProtectionMaxCrashes="5" />
```

If a pool's worker process crashes repeatedly — five times within five minutes, by default — IIS doesn't keep restarting it indefinitely. It takes the **entire pool offline**, returning a `503 Service Unavailable` to every request against that pool, until manually restarted. This is a deliberate protective circuit breaker against a crash-looping application consuming server resources forever, not a bug — but it's a genuinely common source of confusion during troubleshooting, since the symptom (a sudden, total 503 outage) looks exactly like the server itself failing rather than one specific application repeatedly crashing.

Idle Timeout and Always Running Mode

```
<processModel idleTimeout="00:20:00" />
<!-- applicationPool element, IIS 8+ -->
<add name="MyAppPool" startMode="AlwaysRunning" />
```

By default, a pool with no incoming requests for 20 minutes has its worker process shut down entirely, to free up server resources. The next request after that triggers a full cold start — application initialization, and for a .NET app, JIT compilation of the code paths that request touches — producing a real, sometimes multi-second latency spike for whichever visitor happens to make that first request. `startMode="AlwaysRunning"` (paired with the Application Initialization module) keeps the pool's worker process running continuously regardless of traffic gaps, trading a small amount of always-on resource usage for eliminating that cold-start penalty entirely — worth enabling for any site with bursty or low-traffic periods where that first-request latency spike would actually be noticed.

SETTING	CONTROLS	REAL DEFAULT
Identity	Security context of the worker process	ApplicationPoolIdentity
Regular Time Interval	Scheduled recycle	29 hours (1740 minutes)
Rapid-Fail Protection	Circuit breaker on repeated crashes	5 crashes / 5 minutes → pool offline
Idle Timeout	Shuts down an idle worker process	20 minutes

WHERE THIS POINTS FORWARD IN THIS COURSE

A 503 from Rapid-Fail Protection tells you a pool went offline — it doesn't tell you why the application kept crashing in the first place. Chapter 9's **Failed Request Tracing** is the tool that actually captures what was happening inside a request right before a crash, turning "the pool is down" into an actual root cause.

A SUDDEN SITE-WIDE 503 ISN'T ALWAYS A SERVER OUTAGE

It's a very reasonable first assumption that a site suddenly returning 503 to every visitor means the server itself is overloaded or down. If only one specific Application Pool is affected while others on the same server keep responding normally, Rapid-Fail Protection having tripped is a far more likely explanation than genuine server failure — worth checking the Application Pool's own status in IIS Manager (shown as "Stopped") before assuming a much bigger problem.

Hands-On Exercises

EXERCISE 1

An Application Pool's worker process crashes 6 times within a 4-minute window, with rapidFailProtectionMaxCrashes left at its default of 5 and rapidFailProtectionInterval left at its default of 5 minutes. Will Rapid-Fail Protection trigger? Explain using the actual numbers.

View solution

EXERCISE 2

A low-traffic internal site sits idle for over an hour overnight, and the first employee to use it each morning consistently reports the page taking several seconds to load, while every request after that is fast. Explain what's actually happening by default, and what change would fix it.

View solution

EXERCISE 3

Explain, in your own words, why ApplicationPoolIdentity is recommended over a shared built-in identity like NetworkService when a server runs multiple Application Pools for different applications.

View solution

CHAPTER 2 QUICK REFERENCE

- **ApplicationPoolIdentity** — unique virtual account per pool, the modern default for real isolation over shared accounts like NetworkService
- **Recycling** — 29-hour default scheduled interval; memory/request limits off by default; overlapped by default, so a scheduled recycle usually isn't visible to users
- **Rapid-Fail Protection** — 5 crashes in 5 minutes (default) takes the whole pool offline with 503s, a deliberate circuit breaker, not a bug
- **Idle Timeout** — 20 minutes (default) shuts down an idle worker process; `startMode="AlwaysRunning"` avoids the resulting cold-start latency spike

Web Servers Fundamentals Chapter 3 introduced the shape of IIS's own configuration model — `applicationHost.config` machine-wide, `web.config` per-directory — and noted it sits structurally closer to Apache's own `.htaccess` philosophy than to Nginx's single centralized file. What it didn't cover is how a setting actually gets resolved once more than one `web.config` exists in the same directory chain, or what stops a subdirectory from overriding something the machine administrator deliberately locked down. That's this chapter.

The Configuration Hierarchy

Every configurable setting in IIS lives somewhere in a layered hierarchy of XML files, read in a fixed order from the most general to the most specific:

```
%windir%\System32\inetsrv\config\applicationHost.config -- machine-wide, all sites
C:\inetpub\wwwroot\web.config -- site root
C:\inetpub\wwwroot\api\web.config -- /api subdirectory
C:\inetpub\wwwroot\api\admin\web.config -- /api/admin subdirectory
```

A request to `/api/admin/users` doesn't just read the deepest `web.config` in isolation — IIS reads *every* file in the chain above it, starting at `applicationHost.config` and walking down through each `web.config` in turn, merging settings as it goes. This is precisely the same directory-context idea Apache In Depth Chapter 3 covers for `.htaccess` — the deepest applicable file wins for any setting it actually specifies, and any setting it doesn't specify simply falls through to whatever the level above it already resolved to.

How Merging Actually Works: Add, Remove, Clear

For simple scalar settings — a boolean, a numeric limit — a more specific `web.config` value straightforwardly replaces the less specific one. Collection-based settings (a list of default documents, a list of MIME types, a list of modules) behave differently, because IIS's configuration schema exposes three explicit collection actions rather than silently overwriting the whole list:

```
<system.webServer>
<defaultDocument>
<files>
<clear />
<add value="index.html" />
<add value="default.aspx" />
</files>
</defaultDocument>
</system.webServer>
```

`<add>` appends an item to whatever the collection already inherited from the level above. `<remove>` takes out one specific inherited item by name, leaving the rest. `<clear>` discards everything inherited so far and starts the collection fresh at that point in the hierarchy — used here before adding this directory's own two default documents, so this subdirectory's list isn't quietly appended onto whatever the site root or machine level already had configured.

> **FORGETTING <CLEAR /> DOESN'T RESET A COLLECTION — IT APPENDS TO IT**
>
> A very common mistake: adding a `web.config` in a subdirectory intending to *replace* the default-document list, without a `<clear />` first. Without it, the new `<add>` entries land on top of everything already inherited from the site root and machine level, producing a longer, unintended combined list rather than the shorter, intentional one — this is a genuine, easy-to-miss gotcha specific to how IIS's collection inheritance works, with no equivalent concept in a scalar setting.

Locking Sections: overrideModeDefault and allowOverride

Inheritance alone doesn't stop a subdirectory from overriding something a machine administrator never wanted overridden at all — that's a separate, deliberate lock, configured at the machine level:

```
<!-- applicationHost.config -->
<section name="authentication" overrideModeDefault="Deny" />
<section name="defaultDocument" overrideModeDefault="Allow" />
```

Each configuration section — `system.webServer/authentication`, `system.webServer/defaultDocument`, and so on — carries its own `overrideModeDefault`, set in `applicationHost.config`'s own `<sections>` declarations. `Allow` means any `web.config` further down the hierarchy is free to override that section. `Deny` means it isn't — a `web.config` that attempts to set a locked section produces an actual **HTTP 500.19** configuration error rather than silently being ignored. Several security-sensitive sections ship `Deny` by default precisely so that an application deployed by someone without machine-level access can't quietly re-enable something an administrator locked down — `system.webServer/authentication` among them, which is exactly why Chapter 8's Windows Authentication material has to be configured at a level with the necessary permission, not casually from an arbitrary application's own `web.config`.

OVERRIDEMODEDEFAULT	EFFECT	TYPICAL USE
Allow	Any web.config down the chain may override this section	Most content/display sections — defaultDocument, staticContent, etc.
Deny	Only applicationHost.config (or a location tag there) may set it; a web.config attempt fails with 500.19	Security-sensitive sections — authentication, authorization, and similar

Overriding a Locked Section From the Machine Level: <location>

A locked (`Deny`) section isn't necessarily fixed identically for every site forever — an administrator with access to `applicationHost.config` itself can still grant one specific site or path a different value, using a `<location>` tag:

```
<!-- applicationHost.config -->
<location path="Default Web Site/api">
<system.webServer>
<authentication>
<windowsAuthentication enabled="true" />
</authentication>
</system.webServer>
</location>
```

This is how a locked, `Deny`-mode section still ends up different for one specific site or subdirectory without unlocking it for every application on the server — the override lives entirely inside `applicationHost.config`, scoped by `path`, rather than inside that site's own `web.config`. The distinction that matters: a `<location>` block is still machine-level configuration, written by whoever has access to `applicationHost.config` itself — it is not a loophole an application deployer can reach from their own `web.config`.

Resolution Order, Put Together

1. IIS Manager or a config tool checks `applicationHost.config`'s section declarations for `overrideModeDefault` (and any `<location>` override) — this determines whether a lower-level `web.config` is even permitted to touch that section
2. Reading proceeds top-down: `applicationHost.config`, then each `web.config` from the site root down to the requested directory
3. Scalar settings from a deeper file simply replace shallower ones
4. Collections accumulate via `<add>`/`<remove>` unless a `<clear />` resets them at that level
5. The final, fully-merged configuration for that specific request path is what IIS actually applies

WHERE THIS POINTS FORWARD IN THIS COURSE

Chapter 5's virtual directories and application boundaries determine exactly which physical folder a URL path maps onto in the first place — this chapter's own hierarchy walk only makes sense once that mapping is settled. And Chapter 6's URL Rewrite rules are themselves just another `system.webServer` section, subject to the exact same `Allow`/`Deny`/`<location>` mechanics covered here.

A 500.19 ERROR MEANS A LOCKED SECTION, NOT A SYNTAX ERROR

An HTTP 500.19 is easy to mistake for malformed XML, and it's worth checking for that first — but a perfectly well-formed `web.config` that attempts to set a section locked with `overrideModeDefault="Deny"` produces exactly the same 500.19 status. The actual IIS error page names the specific config source and section responsible, which is the fastest way to tell the two apart rather than guessing.

Hands-On Exercises

EXERCISE 1

A site's root web.config sets a defaultDocument list of index.html and default.aspx. A subdirectory's own web.config adds home.html with <add value="home.html" />, but includes no <clear />. List, in order, the final default-document list IIS will actually try for a request into that subdirectory, and explain why.

View solution

EXERCISE 2

A developer with no access to applicationHost.config tries to enable Windows Authentication from their application's own web.config and gets an HTTP 500.19 error. Explain what's actually happening, and describe the one legitimate way an administrator could still grant that specific application a different authentication setting than the rest of the server.

View solution

EXERCISE 3

Explain, in your own words, why IIS's configuration model needs three distinct collection actions (add, remove, clear) instead of just letting a more specific web.config's collection value silently replace a less specific one, the way a scalar setting does.

View solution

CHAPTER 3 QUICK REFERENCE

- **Hierarchy** — applicationHost.config (machine) → web.config at the site root → web.config at each subdirectory down to the request path
- **Scalars** replace; **collections** accumulate via <add>/<remove> unless reset with <clear />
- **overrideModeDefault** — Allow lets a web.config override a section; Deny restricts it to applicationHost.config, producing a 500.19 if violated
- **<location path="...">** — how a machine administrator grants one specific site/path a different value for an otherwise-locked section, without unlocking it server-wide
- A 500.19 can mean either malformed XML or a locked section — the IIS error page names which

Web Servers Fundamentals never covered this at all — the module pipeline is a genuinely new topic for this course, not a deepening of something already introduced. It's also the mechanism that makes Chapter 3's own configuration sections actually do something: a locked `system.webServer/authentication` section only matters because an authentication *module* reads it on every single request, at a specific, predictable point in a fixed processing pipeline.

Request Processing: A Sequence of Named Events

Every request IIS handles moves through a fixed sequence of named pipeline events, in this order:

```
BeginRequest
AuthenticateRequest
AuthorizeRequest
ResolveRequestCache
MapRequestHandler
AcquireRequestState
PreExecuteRequestHandler
ExecuteRequestHandler
ReleaseRequestState
UpdateRequestCache
EndRequest
```

A module doesn't run "somewhere during the request" — it registers itself against one or more of these specific named events, and IIS calls it at exactly that point for every applicable request. An authentication module hooks `AuthenticateRequest`. An authorization module hooks `AuthorizeRequest`, immediately after. Output caching hooks `ResolveRequestCache` (checking for a cached response) and `UpdateRequestCache` (storing a new one) — the exact mechanism behind the Output Caching that Web Servers Fundamentals Chapter 9 only named in passing. The actual content-generating work — running an ASP.NET page, serving a static file — happens at `ExecuteRequestHandler`, which is where **handlers** (below) come in, distinct from modules.

Modules vs. Handlers: A Genuine Distinction

It's easy to conflate the two, but they do different jobs. A **module** can run on *every* request, hooks one or more pipeline events, and typically doesn't generate the actual response body — authentication, authorization, logging, URL rewriting, and compression are all modules. A **handler** is what actually produces the response content for a specific request, and exactly one handler runs per request, selected by matching the request against a table of path/verb patterns:

```
<system.webServer>
<handlers>
<add name="StaticFile" path="*" verb="GET,HEAD"
         modules="StaticFileModule" resourceType="File" />
<add name="AspNetCore" path="*" verb="*"
         modules="AspNetCoreModuleV2" resourceType="Unspecified" />
</handlers>
</system.webServer>
```

A request for `/images/logo.png` matches the `StaticFile` handler mapping and gets served directly off disk. A request into an ASP.NET Core app matches the `AspNetCore` handler mapping and gets handed off to that runtime entirely. Handler mappings are themselves just another `system.webServer` section — subject to exactly the same inheritance, `<clear />` / `<add>` / `<remove>`, and `overrideModeDefault` locking behavior Chapter 3 already covered.

Native Modules vs. Managed Modules

A module is either **native** (a compiled C++ DLL, running as part of the IIS worker process itself — `HttpModule` entries registered in `<globalModules>`) or **managed** (a .NET class implementing `IHttpModule`, running inside the .NET runtime hosted by that same worker process). Both kinds can hook the same pipeline events shown above — the distinction is about what actually executes the module's code, not which events are available to it.

	NATIVE MODULES	MANAGED MODULES
Written in	C++, compiled to a DLL	.NET (C#/VB), an IHttpModule class
Runs	Directly in the IIS worker process	Inside the .NET runtime hosted by the worker process
Applies to	Every request, regardless of content type	Every request, in Integrated mode (see below)
Examples	StaticFileModule, DefaultDocumentModule, HttpCacheModule	FormsAuthenticationModule, SessionStateModule, custom ASP.NET modules

Why Integrated Mode Was a Real Architectural Change

Before IIS 7, a request destined for ASP.NET ran through two genuinely separate pipelines: the native IIS pipeline handled the request first, then handed off to a completely separate ASP.NET pipeline (via the ISAPI extension `aspnet_isapi.dll`) for anything `.aspx`. A managed `IHttpModule` could only ever see the second, ASP.NET-only pipeline — it had no visibility into, say, a plain static file request, since that never entered the ASP.NET pipeline at all. **Integrated mode**, introduced in IIS 7 and the modern default ever since, merges these into one single pipeline: the same eleven events shown above apply uniformly, and a managed module can now hook `AuthenticateRequest` for *any* request — a static image, a classic ASP page, anything — not just ASP.NET content. **Classic mode** (the old two-pipeline behavior) still exists for legacy application compatibility, but is genuinely a compatibility fallback at this point, not a real architectural choice for anything new.

> **WHERE THIS CONNECTS TO THE REST OF THIS COURSE**
>
> Chapter 6's URL Rewrite module and Chapter 8's Request Filtering are both native modules hooking early pipeline events (`BeginRequest` / `ResolveRequestCache`) — which is exactly why a rewrite rule can redirect a request before any handler, native or managed, ever runs against it.

> **"IT'S NOT IN WEB.CONFIG" DOESN'T MEAN "IT ISN'T RUNNING"**
>
> A module can be registered globally in `applicationHost.config`'s `<globalModules>` / `<modules>` sections and simply never appear in any individual application's own `web.config` at all — it still runs against every request that application serves. When troubleshooting unexpected behavior (a header appearing that no application code sets, a response being compressed or cached unexpectedly), checking the full registered module list at the machine level is often more informative than searching application-level configuration for an explanation that isn't there.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, the difference between a module and a handler in IIS's pipeline model, and why exactly one handler runs per request while many modules can run against the same request.

View solution

EXERCISE 2

Before IIS 7's Integrated mode, why couldn't a managed IHttpModule affect how IIS served a plain static image file, even if that module hooked an event equivalent to AuthenticateRequest? What specifically changed with Integrated mode to fix this?

View solution

EXERCISE 3

A site is unexpectedly compressing responses even though no application code or web.config on that site enables compression. Using this chapter's own module concepts, explain the most likely cause and where you'd look to confirm it.

View solution

CHAPTER 4 QUICK REFERENCE

- **Pipeline events** — a fixed sequence (BeginRequest → AuthenticateRequest → AuthorizeRequest → ResolveRequestCache → MapRequestHandler → ... → ExecuteRequestHandler → ... → EndRequest) that every request moves through in order
- **Modules** — hook one or more pipeline events, can run on every request, don't generate the response body themselves (auth, logging, rewriting, compression)
- **Handlers** — exactly one per request, matched by path/verb, actually produce the response content at ExecuteRequestHandler
- **Native** (C++, in-process) vs. **Managed** (.NET, IHttpModule) — both hook the same events, differing in what executes the code
- **Integrated mode** (modern default) merges the old dual native/ASP.NET pipelines into one, letting managed modules see every request; **Classic mode** is a legacy compatibility fallback

Web Servers Fundamentals Chapter 4 covered Sites and bindings at a basic level — enough to know that a binding maps an IP address, port, and host header to a specific site. What it didn't cover is what happens *inside* a site: how a URL path segment gets mapped onto a physical folder, and — more importantly — the real, easy-to-miss distinction between a folder that's merely a virtual directory and one that's been marked an **application**, which is what actually determines Application Pool assignment and .NET runtime isolation.

The Physical-to-Virtual Mapping

```
C:\inetpub\wwwroot\                -- physical path for /
C:\inetpub\wwwroot\images\         -- physical path for /images (plain subfolder, same
app)
D:\shared\assets\                  -- physical path for /static (virtual directory,
different disk)
C:\apps\reporting-api\             -- physical path for /api (a separate application)
```

A site's URL namespace doesn't have to mirror its physical folder layout at all. `/images` above is a plain subfolder that happens to sit directly under the site's own physical root — nothing special about it. `/static` is a genuine **virtual directory**: a URL path segment mapped onto a physical location that isn't even under the site's own root, potentially on an entirely different disk or a UNC network share. Visitors requesting `/static/logo.png` have no way to tell, and don't need to, that the actual file lives on `D:` rather than under `C:\inetpub\wwwroot`.

Virtual Directory vs. Application: The Distinction That Actually Matters

A virtual directory and an application both map a URL path onto a physical folder — the visible URL structure looks identical either way. The difference is invisible from the URL, but very real underneath:

	VIRTUAL DIRECTORY	APPLICATION
Application Pool	Always inherits the parent's pool — no assignment of its own	Has its own Application Pool assignment, independent of the parent
.NET AppDomain	Shares the parent's AppDomain entirely	Gets its own isolated AppDomain, even when assigned to the same pool as the parent
Its own web.config as an app root	Just another web.config in the inheritance chain (Chapter 3)	Same inheritance rules apply, but this is also where a distinct .NET application starts — its own Global.asax, its own compiled assemblies
Created with	`appcmd add vdir`	`appcmd add app`

```
appcmd add vdir /app.name:"Default Web Site/" /path:/static
/physicalPath:"D:\shared\assets"

appcmd add app /site.name:"Default Web Site" /path:/api
/physicalPath:"C:\apps\reporting-api"
appcmd set app "Default Web Site/api" /applicationPool:"ReportingApiPool"
```

This is exactly why Chapter 2's Application Pool identities and recycling schedules are configured *per application*, not per site: a single site can host several genuinely independent applications, each with its own pool, its own worker process, its own crash isolation via Rapid-Fail Protection, and its own recycling schedule — all reachable under one site's bindings.

Why Application Isolation Exists

Marking a folder as an application gives it real isolation a plain virtual directory can't offer. If `/api` above shares an Application Pool with a completely different, unrelated application under the same site, a memory leak or a crash-looping bug in one can bring down the other — Rapid-Fail Protection (Chapter 2) takes the entire pool offline, not just the misbehaving path. Converting `/api` into its own application with its own pool means a crash there produces a 503 scoped to `/api` alone, while the rest of the site keeps responding normally. This is the practical reason a real production deployment often ends up with several small applications under one site rather than one large one — deliberate blast-radius control, not just organizational convenience.

A Real Gotcha: Session State Doesn't Cross Application Boundaries

In-process ASP.NET session state lives inside a specific application's own AppDomain. Converting a subfolder into its own application — precisely to gain the isolation benefit just described — means it can no longer see session state set by its former parent, even though the URL structure looks unchanged to visitors. A folder that was working fine as a plain virtual directory sharing the parent's session state can break in a subtle, hard-to-diagnose way immediately after being promoted to an application, if any code depended on that shared session state existing.

> **WHERE THIS POINTS FORWARD IN THIS COURSE**
>
> Chapter 9's Failed Request Tracing is scoped per application — when troubleshooting, confirming which application boundary a failing request actually falls inside (via this chapter's own physical-path mapping) is the first step, before tracing rules can even be applied correctly.

> **A SUBFOLDER "JUST WORKING" DOESN'T MEAN IT ISN'T ALREADY ISOLATED**
>
> IIS Manager visually nests a virtual directory and an application identically in its tree view — both simply appear as a folder icon under a site. The only reliable way to tell which one you're looking at is checking whether it has its own Application Pool assignment (an application does; a virtual directory never does) — don't assume from the tree view alone.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, the two concrete differences between a virtual directory and an application in IIS, and why converting a folder to an application is the only way to give it its own Application Pool.

View solution

EXERCISE 2

A site hosts two features, /shop and /support, both originally plain subfolders sharing the site's single Application Pool. After a crash in /support repeatedly took down /shop as well, an administrator converts /support into its own application with its own pool. Using Chapter 2's Rapid-Fail Protection material together with this chapter's own isolation concepts, explain why this fixes the cross-impact problem.

View solution

EXERCISE 3

A subfolder that previously worked as a plain virtual directory starts throwing errors related to missing session data immediately after being converted into its own application, with no other code changes made. Explain what most likely happened.

View solution

CHAPTER 5 QUICK REFERENCE

- **Virtual directory** — maps a URL path onto any physical location; always shares the parent's Application Pool and .NET AppDomain
- **Application** — same URL-to-physical-path mapping, but gets its own Application Pool assignment and its own isolated .NET AppDomain
- IIS Manager's tree view looks identical for both — check the Application Pool assignment to tell them apart, not the tree structure
- Application boundaries are deliberate **blast-radius control** — a crash in one application's pool doesn't take down a sibling application's pool under the same site
- **Real gotcha:** in-process session state doesn't cross application boundaries — converting a folder to an application can silently break code relying on shared session state with its former parent

Chapter 1 drew a deliberate line: this course has no dedicated reverse-proxy chapter, because Application Request Routing (ARR) is a genuinely optional add-on most IIS installations never install. URL Rewrite is a separate module from ARR, extremely commonly installed on its own, and doesn't require proxying anything anywhere — it works entirely within a single IIS installation, rewriting or redirecting requests before a handler ever runs. This chapter goes deep on exactly that standalone use.

Rewrite vs. Redirect: Not the Same Action

These two get used almost interchangeably in casual conversation, but they're genuinely different HTTP-level behaviors, and the module treats them as two distinct action types for exactly that reason:

	REWRITE	REDIRECT
What the client sees	Original URL, unchanged in the address bar	A 301/302 response, browser navigates to the new URL
Round trips	One — IIS handles it internally	Two — the client must make a second request
Typical use	Friendly URLs mapping to an actual query-string-based page, internal routing	Enforcing HTTPS, canonical hostname, permanently moved content

Rule Anatomy: Match, Conditions, Action

```
<rewrite>
<rules>
<rule name="Enforce HTTPS" stopProcessing="true">
<match url="(.*)" />
<conditions>
<add input="{HTTPS}" pattern="^OFF$" />
</conditions>
<action type="Redirect" url="https://{HTTP_HOST}/{R:1}"
              redirectType="Permanent" />
</rule>
</rules>
</rewrite>
```

`match url` is a regular expression tested against the request's relative URL. `{R:1}` back-references the first parenthesized capture group from that match — a friendly URL like `/product/42` matched against `^product/([0-9]+)$` can rewrite to `/product.aspx?id={R:1}`, with `{R:1}` substituting the captured `42`. `<conditions>` adds further tests beyond the URL itself — server variables like `{HTTPS}`, `{HTTP_HOST}`, `{HTTP_USER_AGENT}`, or the query string — evaluated with an implicit logical AND by default (`logicalGrouping="MatchAll"`), or OR if set explicitly. `stopProcessing="true"` means: once this rule matches and its action runs, skip every rule listed after it, rather than continuing to test the (already-rewritten) URL against the rest of the rule list.

Rewrite Maps: Lookup Tables Instead of Long Rule Chains

```
<rewriteMaps>
<rewriteMap name="LegacyRedirects">
<add key="/old-about" value="/about-us" />
<add key="/old-contact" value="/contact" />
</rewriteMap>
</rewriteMaps>
<rule name="Legacy URL Map">
<match url="(.*)" />
<conditions>
<add input="{LegacyRedirects:{REQUEST_URI}}" pattern="(.+)" />
</conditions>
<action type="Redirect" url="{C:1}" />
</rule>
```

A single rewrite rule referencing a map like `{LegacyRedirects:{REQUEST_URI}}` replaces what would otherwise be dozens of nearly-identical individual rules, one per legacy URL — genuinely useful after a site restructure with many one-off redirects to preserve. `{C:1}` here back-references the condition's own capture group, distinct from `{R:1}`, which always refers to the `<match>` element's capture instead.

Outbound Rules: Rewriting the Response, Not the Request

Every rule shown so far is an **inbound** rule — it inspects and rewrites the incoming request before a handler runs. An **outbound** rule does the reverse: it inspects and modifies the generated response — most commonly, rewriting hardcoded links inside HTML content so they match a rewritten public URL structure rather than the internal one the application actually generated. Outbound rules require enabling response content rewriting explicitly and carry a genuine performance cost, since IIS has to buffer and scan the full response body rather than passing it straight through — worth reaching for only when a real link-rewriting need exists, not by default.

> **WHERE URL REWRITE DELIBERATELY STOPS, IN THIS COURSE**
>
> The action type `Rewrite` pointed at a different URL entirely (`url="http://backend-server/{R:1}"`) is precisely how URL Rewrite gets combined with ARR to build a reverse proxy — but that combination is exactly the capability Chapter 1 named as out of scope for this course. Everything in this chapter rewrites or redirects *within* the same IIS installation; sending a request to a genuinely different backend server is ARR's job, not covered here.

> **A REWRITE RULE CAN CREATE AN INFINITE LOOP**
>
> If a rule's own output URL happens to match that same rule's `match url` pattern again, IIS re-evaluates the rule against its own rewritten output — and can loop. The module detects this and fails the request with an **HTTP 500.50** error rather than actually looping forever, but tracking down which rule caused it requires checking that a rule's rewritten output can never itself satisfy its own match pattern, particularly with broad patterns like `(.*)`.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, the genuine behavioral difference between a Rewrite action and a Redirect action, including what the client sees and how many HTTP round trips each involves.

View solution

EXERCISE 2

A site needs friendly URLs like /article/hello-world to map internally to /article.aspx?slug=hello-world. Write the match pattern and action needed, and explain what {R:1} refers to in your rule.

View solution

EXERCISE 3

Explain why URL Rewrite pointing an action at a different backend server's URL is exactly the capability this course deliberately excludes, per Chapter 1's own scope reasoning, even though the rule syntax itself looks almost identical to a same-server rewrite.

View solution

CHAPTER 6 QUICK REFERENCE

- **Rewrite** — internal only, URL unchanged in the browser, one round trip; **Redirect** — a real 301/302, browser navigates, two round trips
- Rule anatomy: `<match url>` (regex against the relative URL) → `<conditions>` (server variables, ANDed by default) → `<action>`
- `{R:1}` back-references the match's own capture group; `{C:1}` back-references a condition's capture group — not interchangeable
- **Rewrite maps** replace many near-duplicate rules with one rule plus a lookup table — ideal for bulk legacy-URL redirects
- **Outbound rules** rewrite the response body (e.g. links in HTML), not the request — genuinely different from inbound rules and carries a real performance cost
- A rule whose own output matches its own pattern again can loop — IIS fails with **500.50** rather than looping forever
- Combining Rewrite with a different backend server's URL is ARR's reverse-proxy territory — deliberately out of scope here (Chapter 1)

Web Servers Fundamentals Chapter 7 covered TLS termination generically across all three servers — certificates, chain of trust, the handshake itself. What it didn't cover is that IIS's actual certificate model is genuinely different from Apache's and Nginx's: neither a loose `.pem` file nor a plaintext private key sitting in a config directory, but an entry in the **Windows Certificate Store**, referenced by a binding rather than a file path.

The Windows Certificate Store, Not a File Path

Apache's `SSLCertificateFile` / `SSLCertificateKeyFile` and Nginx's `ssl_certificate` / `ssl_certificate_key` both point directly at files on disk. IIS instead expects a certificate to already be **installed** into the Windows Certificate Store — typically the Local Computer's Personal store (`certlm.msc`, or the `Cert:\LocalMachine\My` PowerShell path) — after which it's identified purely by its **thumbprint**, a SHA-1 hash uniquely fingerprinting that specific certificate:

```
Get-ChildItem -Path Cert:\LocalMachine\My

# Thumbprint                                Subject
# ----------                                -------
# A909502DD82AE41433E6F83886B00D4277A32A7B   CN=www.example.com
```

An HTTPS binding then references that thumbprint rather than a file path — the private key never sits in a plaintext file anywhere IIS's own configuration touches, since the Windows Certificate Store manages the key material itself, with its own export/access permissions independent of `web.config` or `applicationHost.config` entirely.

HTTPS Bindings and SNI

```
netsh http add sslcert hostnameport=www.example.com:443 `
  certhash=a909502dd82ae41433e6f83886b00d4277a32a7b `
  certstorename=MY appid={4dc3e181-e14b-4a21-b022-59fc669b0914} `
  sslctlstorename=CA
```

Server Name Indication (SNI) is what makes it possible to bind several different certificates to the same IP address and port 443, each for a different hostname — the client's browser sends the requested hostname in cleartext as part of the TLS handshake itself, before encryption is established, letting IIS pick the matching certificate before the rest of the handshake proceeds. This exact mechanism is what allows dozens of HTTPS sites to share one IP address, the same underlying idea Web Servers Fundamentals Chapter 7 covered generically for name-based HTTPS virtual hosting. Before SNI support (IIS 8 and earlier lacked it entirely), each distinct certificate genuinely needed its own dedicated IP address, since the server had no way to know which hostname was being requested until *after* the handshake had already committed to one certificate.

The binding dialog's **"Require Server Name Indication"** checkbox is a real, consequential choice, not a formality: enabling it means very old clients that predate SNI support (legacy libraries, some old embedded devices) simply cannot connect at all, since they never send a hostname during the handshake for IIS to match against. Leaving it unchecked keeps a single "default" certificate for connections that don't specify SNI, at the cost of not being able to host multiple independent certificates cleanly on that same IP/port combination.

The Centralized Certificate Store (CCS): A Different Feature Entirely

It's easy to conflate this with the Windows Certificate Store just covered, because the name is so similar — but IIS's **Centralized Certificate Store (CCS)** is a genuinely separate, optional feature, aimed specifically at large-scale multi-tenant hosting with many sites and certificates. Instead of installing every certificate individually into the Windows Certificate Store and configuring a binding referencing its thumbprint, CCS stores certificates as `.pfx` files in one shared folder (often a UNC network path, shared across a web farm), named by convention after the hostname they belong to:

```
\\fileserver\certstore\
  www.example.com.pfx
  api.example.com.pfx
  shop.example.com.pfx
```

IIS looks up the correct `.pfx` file dynamically at handshake time, based on the SNI hostname, rather than requiring a pre-configured binding entry per certificate — genuinely useful in a hosting environment adding and removing tenant certificates constantly, where manually managing individual store entries and bindings for each one doesn't scale.

	WINDOWS CERTIFICATE STORE	CENTRALIZED CERTIFICATE STORE (CCS)
Where certs live	Installed into the OS-level store (Personal, Local Computer)	.pfx files in a shared folder, often a UNC network path
Referenced by	Thumbprint, in an explicit per-site binding	Filename convention matched against the SNI hostname, at handshake time
Best fit	A handful of sites on one server	Many tenants/certificates, often across a shared web farm

WHERE THIS CONNECTS TO CHAPTER 5

An HTTPS binding is configured at the **site** level, in `applicationHost.config` — not per application, and not in any individual application's own `web.config`. Every application under a site, regardless of its own Application Pool assignment from Chapter 5, is reached through whichever certificate that site's own binding presents.

RENEWING A CERTIFICATE DOESN'T UPDATE THE BINDING AUTOMATICALLY

Renewing a certificate — even one for the exact same hostname — produces a brand-new certificate object with a **new thumbprint**, once installed into the store. The existing HTTPS binding still references the old thumbprint and keeps presenting the old (soon-to-expire, or already-expired) certificate until someone explicitly updates the binding to point at the new thumbprint. This genuinely catches people who assume "renewing" is a drop-in replacement — it installs a new certificate alongside the old one, but does nothing to the binding pointing at the old one.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why IIS references a certificate by thumbprint in a binding rather than by a file path the way Apache and Nginx do, and what that means for where the private key material actually lives.

View solution

EXERCISE 2

A team renews a certificate for www.example.com two weeks before it expires, installs the new certificate into the Windows Certificate Store, and considers the task done. On the expiration date, visitors suddenly start seeing certificate warnings. Explain what almost certainly went wrong and what step was missed.

View solution

EXERCISE 3

Explain, in your own words, why the Centralized Certificate Store (CCS) is a genuinely different feature from the Windows Certificate Store despite the similar name, and describe the kind of hosting environment where CCS's own approach scales better than managing individual bindings per certificate.

View solution

CHAPTER 7 QUICK REFERENCE

- IIS certificates live in the **Windows Certificate Store**, referenced by **thumbprint** — not a file path, unlike Apache/Nginx
- **SNI** lets multiple certificates share one IP:port, matched by the hostname sent (in cleartext) during the TLS handshake, before encryption is established
- Pre-SNI, each certificate needed its own dedicated IP address — no way to know the requested hostname before committing to a certificate
- **"Require SNI"** is a real tradeoff: enabling it excludes pre-SNI legacy clients entirely
- **Centralized Certificate Store (CCS)** — a separate, optional feature for large-scale multi-tenant hosting: .pfx files in a shared folder, matched dynamically by hostname, not individual store entries/bindings
- HTTPS bindings are configured at the **site** level in applicationHost.config, applying to every application under that site regardless of Application Pool (Chapter 5)
- **Renewal gotcha:** a renewed certificate gets a new thumbprint — the existing binding must be manually updated to reference it, or the old certificate keeps being presented

Web Servers Fundamentals Chapter 8 named IP/Domain Restrictions and Basic/Windows Authentication, without real directive depth. This chapter goes deep on all three — and Windows Authentication in particular has two genuine operational gotchas (a Kerberos SPN requirement, and the "double-hop" problem) that catch real deployments often enough to be worth understanding before they show up as a confusing, intermittent failure in production.

Windows Authentication: Two Providers, Not One

```
<security>
<authentication>
<windowsAuthentication enabled="true">
<providers>
<clear />
<add value="Negotiate" />
<add value="NTLM" />
</providers>
</windowsAuthentication>
</authentication>
</security>
```

Negotiate attempts Kerberos first, falling back to NTLM only if Kerberos isn't available. **NTLM** is a genuinely older, weaker challenge-response protocol with no support for delegation (below) and no mutual authentication of the server to the client. Listing both providers, with Negotiate first, is the modern default — but the fallback to NTLM is silent and easy to miss, which is exactly what makes the SPN gotcha below so confusing when it happens.

This section lives under `system.webServer/security/authentication` — the same parent path as the `windowsAuthentication` element Chapter 3 used as its own `overrideModeDefault="Deny"` example. That's not a coincidence: authentication is exactly the kind of security-sensitive section that ships locked to `applicationHost.config` by default, and enabling Windows Authentication for a specific site or application generally requires either a machine-level change or a `<location>` override — not a change from that application's own `web.config`.

The SPN Gotcha: "It Works by IP but Fails by Hostname"

Kerberos requires a **Service Principal Name (SPN)** registered in Active Directory for the exact hostname the client is using to reach the site, bound to the account the Application Pool runs under:

```
setspn -A HTTP/www.example.com DOMAIN\svc-webapp
```

Without a matching SPN, Kerberos authentication for that specific hostname fails — and because Negotiate silently falls back to NTLM, the site keeps working, just under the weaker protocol, with no obvious error telling anyone Kerberos never actually succeeded. This produces a real, genuinely confusing pattern: the site "works" when accessed by the server's raw machine name or IP address (where a default SPN often already exists), but a custom public hostname added later — exactly the kind Chapter 5's virtual directories and Chapter 7's SNI bindings both deal with routinely — was never registered, so Kerberos quietly fails for it while NTLM masks the failure entirely.

The Double-Hop Problem

Windows Authentication happily identifies who's making the request to IIS itself — but by default, that identity **cannot be forwarded** a second hop, to a separate backend server the application then needs to call (a SQL Server on its own machine, a separate API server). NTLM has no delegation mechanism at all. Kerberos *can* support delegation, but only if explicitly configured — the Application Pool's identity account needs to be trusted for delegation in Active Directory, and the target service needs its own registered SPN as well. Without that explicit configuration, a call to the second server either fails authentication entirely or silently falls back to the Application Pool's own service identity rather than the original user's — a distinction that matters a great deal if the second server's own access control was meant to be based on the actual end user, not on the web application's service account.

Request Filtering: A Native Pre-Handler Defense Layer

```
<security>
<requestFiltering>
<requestLimits maxAllowedContentLength="30000000"
                  maxUrl="4096" maxQueryString="2048" />
<fileExtensions>
<add fileExtension=".config" allowed="false" />
</fileExtensions>
<hiddenSegments>
<add segment="bin" />
</hiddenSegments>
</requestFiltering>
</security>
```

Request Filtering is a native module (Chapter 4) hooking one of the earliest pipeline events, rejecting a request outright before any handler or authentication module ever runs against it. `requestLimits` caps request-body size, total URL length, and query-string length — a genuine defense against oversized-upload abuse and certain buffer-related exploits, independent of anything the application code itself does. `fileExtensions` denies requests for specific extensions outright — `.config` here, so a misconfigured static-file handler can never accidentally serve a `web.config`'s own contents to a visitor. `hiddenSegments` denies any request whose path contains a named segment anywhere — `bin` here, so `/app/bin/secret.dll` is blocked regardless of what physically exists there, since compiled application binaries have no legitimate reason to ever be served as a direct HTTP response.

IP and Domain Restrictions: Evaluation Order Matters

```
<security>
<ipSecurity allowUnlisted="false">
<add ipAddress="203.0.113.0" subnetMask="255.255.255.0" allowed="true" />
<add ipAddress="203.0.113.50" allowed="false" />
</ipSecurity>
</security>
```

`allowUnlisted` sets the default action for any client IP that matches none of the explicit entries — `false` here means deny-by-default, an allowlist model. Individual entries are evaluated most-specific-match-wins, not strictly top-to-bottom: a single-IP rule (`203.0.113.50`, denied) takes precedence over a broader subnet rule that also happens to contain it (`203.0.113.0/24`, allowed) — letting one specific address be carved out as an exception to an otherwise-allowed range, or vice versa, without needing the entries listed in any particular order.

WHERE THIS POINTS FORWARD IN THIS COURSE

Chapter 9's Failed Request Tracing can capture exactly which module rejected a request and why — including a Request Filtering denial or an authentication failure — turning "the request just got a 403/401" into a concrete, attributable cause rather than a guess.

"IT WORKS, JUST SLOWER" CAN MEAN KERBEROS SILENTLY FAILED

A Windows Authentication setup that appears to work but "feels wrong" — extra prompts, or authentication that works locally but not from certain client machines — is worth checking for a missing or mismatched SPN before assuming a broader configuration problem. Negotiate's silent fallback to NTLM means Kerberos can be failing constantly with no visible error at all, simply working around it every time via the weaker protocol.

Hands-On Exercises

EXERCISE 1

A site using Windows Authentication with the Negotiate provider works fine when accessed via the server's raw machine name, but authentication behaves oddly when accessed via a new public hostname added last week. Using this chapter's own material, explain the most likely cause and the command that would confirm/fix it.

View solution

EXERCISE 2

An ASP.NET application authenticates users with Windows Authentication, then tries to run a query against a separate SQL Server machine using the authenticated user's own Windows identity, expecting the database to enforce per-user permissions. This consistently fails. Explain what's happening, using this chapter's own terminology.

View solution

EXERCISE 3

An ipSecurity section sets allowUnlisted="false", allows the subnet 203.0.113.0/24, and separately denies the single address 203.0.113.50, which falls inside that subnet. Will a request from 203.0.113.50 be allowed or denied? Explain using the actual evaluation rule.

View solution

CHAPTER 8 QUICK REFERENCE

- **Negotiate** tries Kerberos first, silently falls back to **NTLM** — the fallback is invisible unless specifically checked for
- **SPN gotcha:** Kerberos needs a registered SPN for the exact hostname used; missing it fails Kerberos silently (masked by the NTLM fallback)
- **Double-hop problem:** a user's Windows identity doesn't forward to a second backend server by default — NTLM never supports it; Kerberos needs explicit delegation configuration
- **Request Filtering** — a native module rejecting oversized/malformed/forbidden requests before any handler or auth module runs (size limits, file extensions, hidden path segments)
- **IP/Domain Restrictions** — `allowUnlisted` sets the default for unmatched IPs; specific entries win over broader ones regardless of listed order
- Authentication is a security-sensitive section, typically `Deny`-locked at the machine level (Chapter 3) — expect a `<location>` override or machine-level change, not a per-application `web.config` edit

Web Servers Fundamentals Chapter 9 named Dynamic/Static Content Compression and Output Caching briefly, comparatively. This chapter goes deep on both, on real logging configuration, and — finally — on **Failed Request Tracing**, the tool Chapters 2, 5, and 8 each pointed forward to as the definitive way to turn a confusing failure into an actual root cause.

Static vs. Dynamic Compression, and IIS's Own CPU Throttle

```
<httpCompression dynamicCompressionDisableCpuUsage="90"
                 dynamicCompressionEnableCpuUsage="70">
</httpCompression>
<urlCompression doStaticCompression="true" doDynamicCompression="true" />
```

Static compression compresses a static file once and caches the compressed copy on disk, serving that cached copy for every subsequent request — genuinely cheap, since the CPU cost is paid exactly once no matter how many times the file is requested afterward. **Dynamic compression** compresses generated, per-request content fresh every single time, since the content itself is different on each request — a real, ongoing CPU cost that scales directly with request volume. IIS has a genuinely distinctive safeguard here that neither Apache nor Nginx has built in the same way: `dynamicCompressionDisableCpuUsage` automatically *turns off* dynamic compression once server CPU usage crosses that threshold, and `dynamicCompressionEnableCpuUsage` turns it back on once usage drops below the lower one — an adaptive mechanism that trades away some bandwidth savings under real load rather than letting compression itself become the thing that overloads the server.

Kernel-Mode Output Caching vs. User-Mode Caching

Web Servers Fundamentals' own comparative Output Caching mention didn't distinguish these two — but the difference is substantial. IIS's **kernel-mode cache** lives inside `http.sys`, the kernel driver that receives every incoming HTTP request before it ever reaches user-mode code at all. A cache hit there is served directly by the kernel driver itself — the IIS worker process (`w3wp.exe`) never even wakes up for that request, which is why kernel-mode cache hits are dramatically cheaper than any user-mode alternative. The **user-mode output cache** (`system.webServer/caching`) is the fallback for content that can't be kernel-cached at all — it still requires the worker process to run, just skipping the actual content-generation step on a cache hit.

```
<caching>
<profiles>
<add extension=".aspx" policy="CacheForTimePeriod"
         kernelCachePolicy="CacheForTimePeriod" duration="00:05:00" />
</profiles>
</caching>
```

CERTAIN MODULES SILENTLY DISABLE KERNEL-MODE CACHING FOR THEIR OWN CONTENT

Kernel-mode caching only works for content `http.sys` itself can safely serve without any user-mode involvement — which means any response that could legitimately vary in a way the kernel driver can't evaluate on its own is ineligible. In practice, this includes content going through Chapter 6's URL Rewrite rules (the response depends on rule logic that only runs in user mode) and anything requiring authentication beyond anonymous access (Chapter 8) — Windows Authentication's own per-user response can't be safely served from a single shared kernel-mode cache entry. Content that "should" be cached and isn't, with no error anywhere, is very often explained by one of these two features being active on the same path.

Logging: W3C Extended Format and Rollover

```
<logFile logFormat="W3C" period="Daily"
         logExtFileFlags="Date, Time, ClientIP, UserName, Method, UriStem, UriQuery,
HttpStatus, TimeTaken" />
```

The **W3C Extended Log File Format** is IIS's own default, configurable field-by-field — `time-taken` (request duration in milliseconds) is particularly useful paired with Failed Request Tracing below, since a slow-but-successful request often needs a different diagnosis than an outright failure. Log files land under `%SystemDrive%\inetpub\logs\LogFiles` by default, one subfolder per site, rolling over on the configured `period` (Hourly/Daily/Weekly/Monthly) or once a file reaches `truncateSize`. For a multi-server web farm, **Centralized Binary Logging** is a real alternative worth knowing about: a single shared binary log file across every server rather than separate per-server text logs, avoiding the need to reconcile logs from several machines by hand — a genuinely different tradeoff from either Apache's or Nginx's own typically per-server text log files.

Failed Request Tracing: The Forward-Referenced Payoff

```
<tracing>
<traceFailedRequests>
<add path="*">
<traceAreas>
<add provider="ASP" verbosity="Verbose" />
<add provider="WWW Server" verbosity="Verbose" />
</traceAreas>
<failureDefinitions statusCodes="401,403,500-599" timeTaken="00:00:10" />
</add>
</traceFailedRequests>
</tracing>
```

This is the tool Chapter 2 named as the way to find out *why* a pool kept crash-looping into Rapid-Fail Protection, that Chapter 5 named as scoped per application, and that Chapter 8 named as the way to turn "the request got a 401/403" into an actual cause. `failureDefinitions` tells IIS exactly which requests are worth capturing — here, a specific set of status codes or any request taking over 10 seconds — rather than tracing every single request, which would be prohibitively expensive and fill disk space fast. Once a matching request occurs, IIS writes a detailed XML file recording *every* pipeline event and module notification the request passed through, each with its own precise timestamp — the exact module that returned 401, the exact point a request crossed the 10-second threshold, or the exact point in the pipeline where a worker process crash actually occurred. Opened in a browser (the generated XML ships with its own XSL stylesheet), this turns a bare status code or a Rapid-Fail Protection 503 into a genuine, attributable root cause — module by module, event by event.

> **WHERE THIS CLOSES THE LOOP ON THIS COURSE**
>
> Every forward reference this course has made — Chapter 2's crash-looping pool, Chapter 5's per-application troubleshooting, Chapter 8's authentication/filtering denials — resolves here. Failed Request Tracing is genuinely the diagnostic capstone underneath everything covered so far, which is exactly why the capstone in Chapter 10 uses it as the final verification step for the deployment it builds.

TRACING EVERYTHING, ALWAYS, FILLS A DISK

A `path="*"` rule with broad `failureDefinitions` left enabled indefinitely on a busy production site can generate a very large volume of XML trace files, since every single matching request produces its own file. Scoping `failureDefinitions` tightly (specific status codes, a meaningful `timeTaken` threshold) and disabling the rule once a specific investigation is done is the practical approach — not leaving broad tracing permanently active "just in case."

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why a kernel-mode output cache hit is significantly cheaper than a user-mode output cache hit, in terms of what actually has to run to serve the response.

View solution

EXERCISE 2

A page is configured with a kernel-mode caching profile, but an administrator notices via logging that every single request still reaches the application code — no cache hits are occurring at all. The same site also has a URL Rewrite rule active on that exact path. Explain the most likely reason kernel-mode caching isn't working here.

View solution

EXERCISE 3

Referring back to Chapter 2's Rapid-Fail Protection scenario (a pool repeatedly crashing and going offline), explain specifically what Failed Request Tracing would capture that the bare 503 status code and Chapter 2's own recycling settings never could, and what failureDefinitions configuration would be appropriate to actually capture it.

View solution

CHAPTER 9 QUICK REFERENCE

- **Static compression** — cached once, cheap forever; **Dynamic compression** — recomputed per request, real ongoing CPU cost
- IIS auto-throttles dynamic compression via `dynamicCompressionDisableCpuUsage` / `...EnableCpuUsage` — a genuine adaptive safeguard neither Apache nor Nginx has built in the same way
- **Kernel-mode caching** (http.sys) serves a hit without ever waking the worker process — far cheaper than **user-mode caching**, which still requires the worker process to run
- URL Rewrite rules and non-anonymous authentication both silently disable kernel-mode caching eligibility for affected content
- **W3C Extended logging** — configurable fields, rollover by period or size; **Centralized Binary Logging** for web farms avoids per-server log reconciliation
- **Failed Request Tracing** — scoped by `failureDefinitions` (status codes, time taken), captures every pipeline event/module notification for a matching request as a browsable XML file — the payoff for every troubleshooting forward-reference earlier in this course
- Scope tracing tightly and disable it after use — broad, permanent tracing on a busy site fills disk space fast

Web Servers Fundamentals' own capstone compared three servers across several scenarios. This one is different, by design — the same shape Nginx In Depth's and Apache In Depth's own capstones both used: a single, cohesive worked example, layering in each of this course's own eight prior chapters, one at a time, until one complete, production-shaped IIS deployment exists.

The Scenario

One IIS server hosts one site, `www.example.com`, serving a customer-facing ASP.NET Core storefront application at the site root, plus a small internal reporting API reachable at `/api` — used only by staff on the corporate network, never by customers. Both need to keep working reliably, and the internal API specifically must never be able to take the storefront down with it if something goes wrong.

Step 1 — Application Pool Design (Chapter 2)

```
appcmd add apppool /name:"StorefrontPool"
/processModel.identityType:ApplicationPoolIdentity
appcmd set apppool "StorefrontPool" /startMode:AlwaysRunning

appcmd add apppool /name:"ReportingApiPool"
/processModel.identityType:ApplicationPoolIdentity
appcmd set apppool "ReportingApiPool" /recycling.periodicRestart.time:"1.05:00:00"
```

Two separate pools, each under its own dedicated **ApplicationPoolIdentity** — no shared built-in account, so a compromise or misconfiguration in one pool's identity carries no access to the other's resources. The storefront pool runs `AlwaysRunning`, since checkout traffic can't tolerate a cold-start latency spike. The reporting API pool keeps the default 29-hour scheduled recycle — it's genuinely low-traffic, so a brief overlapped recycle is a non-issue.

Step 2 — Locking the Configuration Model (Chapter 3)

```
<!-- applicationHost.config -->
<location path="Storefront/api">
<system.webServer>
<authentication>
<windowsAuthentication enabled="true" />
</authentication>
</system.webServer>
</location>
```

The `authentication` section ships `Deny`-locked by default, exactly as Chapter 3 covered — so enabling Windows Authentication for `/api` specifically (needed in Step 7) has to happen via a `<location>` block in `applicationHost.config`, not from `/api`'s own `web.config`. Every other section stays at its default `overrideModeDefault`, since neither application needs anything else locked down beyond this one deliberate exception.

Step 3 — Handler Mapping for ASP.NET Core (Chapter 4)

```
<system.webServer>
<handlers>
<add name="aspNetCore" path="*" verb="*"
         modules="AspNetCoreModuleV2" resourceType="Unspecified" />
</handlers>
<aspNetCore processPath="dotnet" arguments=".\Storefront.dll" />
</system.webServer>
```

Both applications run in **Integrated mode**, the modern default from Chapter 4 — the storefront's own managed authentication and session modules see every request uniformly, static assets included, exactly the unified pipeline that would have been impossible under the old Classic mode's split native/ASP.NET handling.

Step 4 — Promoting /api to Its Own Application (Chapter 5)

```
appcmd add app /site.name:"Storefront" /path:/api /physicalPath:"C:\apps\reporting-api"
appcmd set app "Storefront/api" /applicationPool:"ReportingApiPool"
```

`/api` is deliberately an **application**, not a plain virtual directory — precisely so it gets its own Application Pool and its own isolated .NET AppDomain, per Chapter 5. If the reporting API ever crash-loops into Rapid-Fail Protection, only `ReportingApiPool` goes offline — the storefront's own `StorefrontPool` is a completely separate worker process, untouched.

Step 5 — URL Rewrite for the Storefront (Chapter 6)

```
<rule name="Enforce HTTPS" stopProcessing="true">
<match url="(.*)" />
<conditions>
<add input="{HTTPS}" pattern="^OFF$" />
</conditions>
<action type="Redirect" url="https://{HTTP_HOST}/{R:1}" redirectType="Permanent" />
</rule>
<rule name="Friendly Product URL">
<match url="^product/([0-9]+)$" />
<action type="Rewrite" url="product.aspx?id={R:1}" />
</rule>
```

Applied only to the storefront, not `/api` — a real customer-facing Redirect to enforce HTTPS, and an internal Rewrite for clean product URLs, exactly the Rewrite-vs-Redirect distinction Chapter 6 drew. No rule here ever points at a different backend server's URL — this stays entirely within-server rewriting, deliberately not reaching for ARR.

Step 6 — HTTPS Binding with SNI (Chapter 7)

```
netsh http add sslcert hostnameport=www.example.com:443 `
  certhash=a909502dd82ae41433e6f83886b00d4277a32a7b `
  certstorename=MY appid={4dc3e181-e14b-4a21-b022-59fc669b0914}
```

A single certificate, installed in the Windows Certificate Store and bound by thumbprint via SNI — `/api` shares this exact binding, since it's a path under the same site, not a separate site with its own certificate needs. A recurring calendar reminder is set for 30 days before expiry, specifically so the binding's thumbprint gets updated the moment the certificate is renewed, per Chapter 7's own renewal gotcha.

Step 7 — Authentication and Filtering for /api (Chapter 8)

```
setspn -A HTTP/www.example.com DOMAIN\svc-reporting

<security>
<ipSecurity allowUnlisted="false">
<add ipAddress="10.0.0.0" subnetMask="255.0.0.0" allowed="true" />
</ipSecurity>
<requestFiltering>
<fileExtensions>
<add fileExtension=".config" allowed="false" />
</fileExtensions>
</requestFiltering>
</security>
```

`/api` uses Windows Authentication (enabled via Step 2's own `<location>` override) with an SPN explicitly registered for `www.example.com` against the reporting service account — done up front here specifically to avoid Chapter 8's own silent-NTLM-fallback trap. `ipSecurity` restricts `/api` to the corporate network's own subnet, deny-by-default. Request Filtering blocks `.config` requests site-wide, so neither application can ever accidentally expose its own `web.config` contents.

Step 8 — Performance, Logging & a Pre-Flight Tracing Check (Chapter 9)

```
<urlCompression doStaticCompression="true" doDynamicCompression="true" />
<caching>
<profiles>
<add extension=".css" policy="CacheForTimePeriod"
         kernelCachePolicy="CacheForTimePeriod" duration="01:00:00" />
</profiles>
</caching>
<tracing>
<traceFailedRequests>
<add path="/api/*">
<failureDefinitions statusCodes="401,403,500-599" />
</add>
</traceFailedRequests>
</tracing>
```

Static assets (`.css`, images) get kernel-mode caching — genuinely eligible, since they're anonymous and untouched by URL Rewrite. The storefront's own product pages stay out of kernel-mode caching deliberately, since Step 5's own Rewrite rule applies to them, exactly the eligibility gotcha Chapter 9 named. Failed Request Tracing is scoped tightly to `/api` alone and only to genuine failure status codes — enabled for the initial rollout window to verify the new Windows Authentication and IP restriction rules actually behave as intended, with a plan to disable it once that's confirmed rather than leaving it running indefinitely.

CAPSTONE STEP	CHAPTER IT DRAWS FROM
Step 1 — Application Pool identities & recycling	Chapter 2
Step 2 — Locking authentication via <location>	Chapter 3
Step 3 — Handler mapping & Integrated mode	Chapter 4
Step 4 — /api as its own isolated application	Chapter 5
Step 5 — URL Rewrite (HTTPS enforcement, friendly URLs)	Chapter 6
Step 6 — HTTPS binding with SNI	Chapter 7
Step 7 — Windows Authentication, SPN, IP restriction, filtering	Chapter 8
Step 8 — Compression, caching, scoped Failed Request Tracing	Chapter 9

THREE CAPSTONES, ONE FULL JOURNEY

Web Servers Fundamentals' own capstone answered "which server fits this job." This one answers "given IIS was already chosen, how do you actually configure it well" — the same relationship Nginx In Depth's and Apache In Depth's own capstones have to that same earlier course. Together, all four capstones across the Web Servers subject cover the full journey from choosing a web server to running a genuinely production-shaped configuration of whichever one was chosen.

THIS DEPLOYMENT IS A SOLID FOUNDATION, NOT A COMPLETE GO-LIVE CHECKLIST

Every setting above reflects real material from this course, but a genuine production rollout still needs steps beyond any single configuration pass: a staged rollout rather than switching all traffic at once, real alerting layered on top of the logging configured here, and a documented certificate-renewal process so Chapter 7's own thumbprint gotcha never becomes an actual outage. Treat this chapter's deployment as a strong, correct starting point, not a substitute for an actual production rollout process.

Hands-On Exercises

EXERCISE 1

Explain why /api was deliberately made its own application in Step 4 rather than left as a plain subfolder under the storefront's own application, referencing both Chapter 2 and Chapter 5's own material.

View solution

EXERCISE 2

A colleague suggests also adding the storefront's own /product.aspx pages to the kernel-mode caching profile used for .css files in Step 8, to speed things up further. Explain why this wouldn't actually work as intended, using this chapter's own Step 5 and Step 8 together.

View solution

EXERCISE 3

Write a short chapter-attribution summary (2-3 sentences) explaining how this capstone's own shape differs from Web Servers Fundamentals' own capstone, and why that difference makes sense given what each course actually covers.

View solution

COURSE COMPLETE

IIS In Depth — 10 of 10 chapters complete. The Web Servers subject now has its comparative foundation, its Nginx deep dive, its Apache deep dive, and its IIS deep dive.

CHAPTER 10 QUICK REFERENCE

- This capstone builds **one cohesive production deployment** for a public storefront and an isolated internal API under the same site, layering in Chapters 2 through 9 step by step — the same deliberate shape as Nginx In Depth's and Apache In Depth's own capstones, and a different one from Web Servers Fundamentals' own multi-scenario decision framework
- Promoting `/api` to its own application (Chapter 5) with its own Application Pool (Chapter 2) is what keeps a reporting-API crash from ever taking the storefront down with it
- Kernel-mode caching only applies to content untouched by URL Rewrite or non-anonymous authentication (Chapter 9) — deliberately excluded here for the storefront's own rewritten product pages
- Registering the SPN and restricting Windows Authentication and IP access to `/api` up front avoids both of Chapter 8's own named gotchas — silent NTLM fallback and an open internal endpoint
- A working deployment is a **foundation**, not a complete production rollout — staged traffic cutover, real alerting, and a documented certificate-renewal process still matter beyond this chapter's own scope