WEB PROGRAMMING

First Edition

ONE MENTAL MODEL. THREE REAL STACKS. WORKING CRUD APPS.

Irfan Subakti

Profitina — profitina.com

TABLE OF CONTENTS

Preface

This book grew out of the Web Programming course I teach in the Department of Informatics at Institut Teknologi Sepuluh Nopember (ITS) Surabaya, and it is organised the way the course itself is organised: 16 chapters, each one a session you could sit through, work through, and then use.

Before a single line of PHP, JSP, or C# is written, this book grounds every chapter in one clear mental model: what actually happens between typing a URL and a page rendering on your screen — then proves that model by building the same CRUD application three separate times, once per stack.

Web Programming (WebPro) opens with the client-server model and the HTTP request-response cycle — DNS lookups, requests, responses — and HTML, CSS and JavaScript fundamentals, closed out with a real take-home website project. It then runs the same core exercise three times over in three different technology stacks: PHP with MySQL (moving from raw PHP to the MVC pattern via Laravel), Java with JSP (fundamentals, a full CRUD walkthrough, and XML/JSON data handling), and ASP.NET Web Forms (GridView, ADO.NET, validation, and state management). Each stack closes with its own small, real take-home project — Quiz 1, a Midterm, Quiz 2, and a Final Exam — so the course is graded on working applications, not multiple-choice recall.

By the time you reach the last page, you should be able to:

- Explain precisely what happens between typing a URL and a rendered page — DNS resolution, the HTTP request-response cycle, client vs. server responsibilities.
- Structure content with HTML and style and script it with CSS and JavaScript.
- Build a working PHP + MySQL CRUD application, and see why an MVC framework like Laravel exists.
- Build the same kind of application again with Java JSP, including working with XML and JSON.
- Build it a third time with ASP.NET Web Forms, using GridView, ADO.NET, validation, and state management.
- Compare three real web stacks by having actually built the same problem in each one, not by reading about them.

The running example in these pages is Profitina Laundry — a small but complete laundry-business application, built once per book, from an empty folder to something that actually runs. It is deliberately modest in scope and deliberately honest in detail: the same validation, the same database access, the same deployment questions a real customer-facing application has to answer.

Who this book is written for: web-programming undergraduates taking their first course in building database-backed web applications, and self-learners who want a practical, from-scratch tour of PHP, JSP, and ASP.NET rather than a single-framework tutorial.

A word on method. Every code example in this book was compiled and run before it was written down, and the appendices carry the verification logs to prove it. Where a number appears, it came from a measurement rather than from memory. If you find an error, or a passage that could be clearer, I would genuinely like to hear about it — that is how the next edition gets better.

Surabaya, September 2026

Irfan Subakti
yifana@gmail.com

Profitina

profitina.com

CHAPTER 1

Foundations of the Web: How a Web Page Actually Gets to Your Screen

Before writing a single line of HTML, PHP, JSP, or C#, every web programmer needs a clear mental model of what actually happens between typing a URL and seeing a page appear.

Learning Objectives — after this chapter you will be able to:

(1) Distinguish the Internet from the Web, and explain what each layer contributes to a page view. (2) Explain the HTTP request-response cycle end to end, and name each of its five steps. (3) Compare the roles of a web browser, a web server, and a database in a typical web application. (4) Trace the historical evolution of the Web from 1.0 through 5.0, and recognize which era a given technology belongs to. (5) Identify the three server-side stacks this course covers — PHP/MySQL, JSP, and ASP.NET/ADO.NET — and explain, at a high level, how each fits into the request-response cycle.

One Toolchain, Three Operating Systems — Windows 11, macOS, Ubuntu

OS	Local web server (PHP)	Editor
Windows 11	XAMPP for Windows (apachefriends.org) or php -S localhost:8000	VS Code + PHP/HTML extensions
macOS	brew install php then php -S localhost:8000	VS Code + PHP/HTML extensions
Ubuntu/Linux	sudo apt install php then php -S localhost:8000	VS Code + PHP/HTML extensions

Every example in this book runs identically on Windows 11, macOS, and Ubuntu/Linux; commands differ only at install time. Pick your row once and follow along on your own laptop.

1.1 Welcome to Web Programming

Welcome to Web Programming (WebPro). This course is not about memorizing the syntax of one language. It is about understanding a single recurring pattern — a client asks, a server answers — and then learning to implement that pattern using three of the most widely deployed server-side technology stacks in the industry: PHP with MySQL, Java Server Pages (JSP) with a Java servlet container, and ASP.NET with ADO.NET. Every technology you will meet in the next fifteen lectures is, underneath its particular syntax, doing exactly the same job: receiving an HTTP request, deciding what it means, fetching or updating data, and sending back a response the browser can render. To keep every idea concrete rather than abstract, this book builds toward one running example you will meet formally in Section 1.7 and again in Lectures 5–16: Profitina Laundry, a real laundry-service business whose booking-and-billing system you will design and then implement three separate times, once per stack.

By the end of this course you will be able to design a relational database for a real business problem, build a complete CRUD (Create, Read, Update, Delete) web application against that database in more than one server-side technology, and explain — precisely, not just by feel — what happens at every stage between a user's click and the screen updating in response.

Why three stacks, not one?

It would be faster to teach only one framework. But real engineering organizations run on a mixture of legacy PHP systems, Java enterprise backends, and .NET services, often all three inside the same company. Learning the underlying request-response pattern once, and then seeing it implemented three different ways, is what lets you pick up a fourth or fifth framework quickly later in your career — because you will already recognize the pattern underneath the new syntax.

1.2 The Internet vs. The Web — Two Different Things

In everyday speech people say “the Internet” and “the Web” interchangeably, but for a web programmer the distinction matters. The Internet is the global physical and logical infrastructure — millions of interconnected routers, cables, satellite links, and the TCP/IP protocol suite that lets any two connected machines exchange packets of data, no matter where in the world they are. The Internet existed, and carried traffic, before the Web did: email, file transfer (FTP), and remote terminal access (Telnet) all ran on the Internet in the 1970s and 1980s.

The Web — short for the World Wide Web — is one particular application that runs on top of the Internet. It was invented in 1989–1991 by Tim Berners-Lee at CERN, and it consists of three ingredients working together: HTML (a document format for structuring content and linking between documents), HTTP (a transfer protocol for requesting and delivering those documents), and URLs (a naming scheme for locating any document, anywhere, uniquely). Every technology covered in this course — HTML, CSS, JavaScript, PHP, JSP, ASP.NET — exists inside this one application layer, running on top of the Internet's lower layers.

A postal analogy

The Internet is like the worldwide postal and road network — the trucks, ships, sorting depots, and addressing conventions that can move an envelope from any city to any other. The Web is like one particular kind of mail — a specific envelope format (HTML), a specific delivery request form you fill out (HTTP), and a specific addressing scheme for that mail only (URLs). Email and file-sharing services are different kinds of "mail" that use the same underlying network but a different envelope format and set of rules.

1.3 The Client-Server Model and the HTTP Request-Response Cycle

Almost everything you build in this course follows the client-server model: a client (typically a web browser) initiates a request, and a server (a program listening for connections, typically on port 80 for plain HTTP or port 443 for encrypted HTTPS) processes that request and sends back a response. This is a request-response pattern — the server never pushes a page to you unprompted; it only ever replies to a request you (or your browser) made first.

The Client-Server Model and the HTTP Request-Response Cycle

Every ordinary web page view is one round trip through this cycle — a request out, a response back.

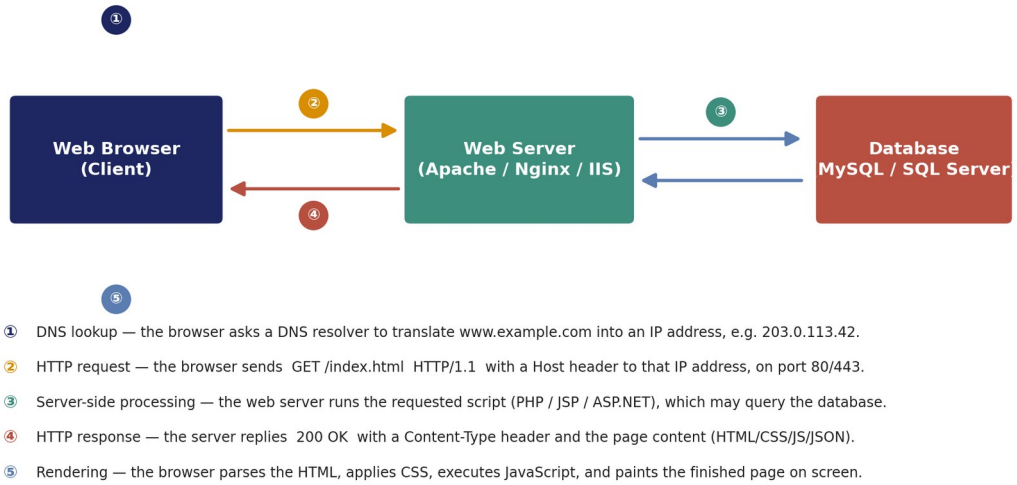

Figure 1.1 — The HTTP request-response cycle: a DNS lookup resolves the domain name, the browser sends an HTTP request, the server (optionally consulting a database) processes it, and an HTTP response is rendered by the browser.

Step ① happens first and separately from the rest: before the browser can even contact www.example.com, the operating system asks a DNS (Domain Name System) resolver to translate that human-readable domain name into a numeric IP address, such as 203.0.113.42 — routers only know how to move packets toward IP addresses, not toward names. Steps ②–④ are the actual HTTP exchange: the browser opens a connection to that IP address and sends a request line such as `GET /index.html HTTP/1.1` together with a `Host:` header (needed because a single IP address commonly hosts many different domains), the server's software decides how to answer — for a static file it simply reads the file from disk, for a dynamic page it runs a PHP, JSP, or ASP.NET script, which may itself query a database — and the server replies with a status line such as `HTTP/1.1 200 OK`, a set of headers (including `Content-Type`, telling the browser how to interpret what follows), and a body containing the actual content. Step ⑤ is entirely the browser's own work: parsing the HTML into a tree of elements (the DOM), applying CSS rules to decide how each element looks, executing any JavaScript, and finally painting pixels on the screen. Every one of these five steps happens, unglamorously, whenever a real customer opens Profitina Laundry's bill-lookup page or Profitina's own marketing site — there is no separate, simpler mechanism reserved for “real” production traffic.

> **HTTP is stateless**
>
> By design, HTTP treats every request as completely independent of every other — the protocol itself has no built-in memory of a previous request from the same browser. This is called being stateless. It is a deliberate simplicity/scalability trade-off: a stateless server can be restarted, load-balanced across many machines, or scaled horizontally without needing to keep track of who is “still connected”, because nobody ever really stays connected between requests. The cost is that any web application needing to remember something about a specific visitor — that they are logged in, or what is in their shopping cart — has to add that memory back deliberately, using cookies and sessions. You will implement exactly this in Lecture 6.

1.3.1 The Most Common HTTP Methods

The request line's first word — the HTTP method — tells the server what kind of action the client wants performed. This course uses four methods repeatedly, and they map naturally onto the four CRUD operations you will implement in every stack.

Method	Typical purpose	CRUD operation	Safe / Idempotent?
GET	Retrieve a resource (a page, a record, a list)	Read	Safe & idempotent
POST	Submit data to create a new resource, or trigger an action	Create	Neither
PUT	Replace a resource entirely with the supplied data	Update	Idempotent, not safe
DELETE	Remove a resource	Delete	Idempotent, not safe

“Safe” means the method must not change server state (a GET request should never delete data), and “idempotent” means sending the identical request twice has the same effect as sending it once (deleting the same record twice leaves the same end state as deleting it once). These are conventions the HTTP standard expects every well-behaved application to respect, not something the protocol enforces automatically — nothing stops a careless program from performing a DELETE inside a GET handler, but doing so breaks caching, breaks browser “back/forward”/prefetch behavior, and is considered a serious design bug.

1.3.2 The Most Common HTTP Status Codes

The response's status code is a three-digit number that tells the client, at a glance, what happened. Status codes are grouped into five families by their first digit.

Code	Meaning	When you will see it
200 OK	Success	Every ordinary successful page load or API call.
301 / 302	Redirect (permanent / temporary)	After a successful login, `header("Location: ...")` in PHP.
400 Bad Request	The client sent malformed data	A form submitted with a required field missing or mistyped.
401 Unauthorized	Authentication is required	Accessing a protected page while not logged in.
403 Forbidden	Authenticated, but not allowed	A logged-in non-admin user requesting an admin-only page.
404 Not Found	No resource exists at this URL	A mistyped URL, or a deleted record's detail page.
500 Internal Server Error	The server-side code crashed	An uncaught PHP/JSP/C# exception while handling the request.

1.4 Web Servers: The Software That Actually Answers Requests

A web server is the software process that listens on a network port, accepts incoming HTTP connections, and decides how to answer each request — by serving a static file directly from disk, or by handing the request off to a language runtime (PHP, a Java servlet container, the ASP.NET pipeline) that produces a dynamic response. This course's three tracks each pair naturally with a specific server: PHP is most commonly served by Apache or Nginx (often bundled together with MySQL and PHP itself into the "XAMPP" or "LAMP/LEMP" development stack you will install in Lecture 5–6); JSP requires a Java servlet container such as Apache Tomcat; and ASP.NET is served by IIS (Internet Information Services) on Windows, or by the cross-platform Kestrel server bundled with modern .NET.

> **Current web server market share (W3Techs, August 2026)**
>
> Across all websites W3Techs surveys, Nginx leads with roughly 31% market share, Cloudflare's own edge server is close behind at roughly 30%, Apache holds about 23%, and LiteSpeed has grown to about 15% (note that many sites report more than one server in their stack, so these figures do not sum to 100%). Microsoft-IIS, this course's ASP.NET server, sits at a modest 3%, reflecting how much of the modern public web is Linux-hosted rather than Windows-hosted — but IIS (and its cross-platform successor, Kestrel) remains the standard inside countless enterprises running internal ASP.NET line-of-business applications, which is exactly the kind of system Lectures 13–15 prepare you to build.

1.5 The Evolution of the Web: From Static Pages to an Intelligent, User-Owned Web

The Web you use today looks nothing like the Web of 1995, even though the underlying HTTP/HTML/URL foundation from Section 1.2 has not fundamentally changed. What has changed is what people build on top of that foundation, and each era is commonly given a version-style label — understanding these labels helps you place any technology you encounter, in this course or afterward, in its proper historical and technical context (Figure 1.2).

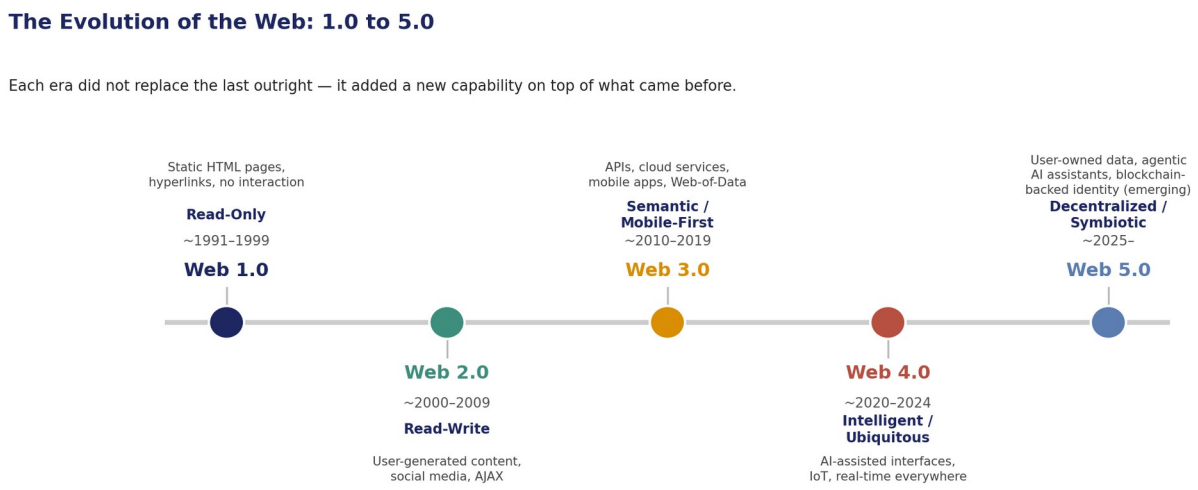

Figure 1.2 — Five recognized eras of the Web. Each era layered new capability on top of the one before it rather than replacing it outright.

Web 1.0 (roughly 1991–1999) was “read-only”: individual authors and organizations published static HTML pages, connected by hyperlinks, with essentially no server-side interactivity — this is the Web your Lecture 2 (plain HTML) content most directly recreates. Web 2.0 (roughly 2000–2009) became “read-write”: technologies like AJAX let JavaScript update part of a page without a full reload, and platforms built around user-generated content — blogs, wikis, and the first social networks — exploded, all of which depends on exactly the server-side processing (PHP, in this course) covered starting in Lecture 3. Web 3.0 (roughly 2010–2019) is usually characterized as the API-driven, mobile-first, semantic Web: applications stopped being single monolithic web pages and became compositions of many services talking to each other over APIs, frequently consumed by native mobile apps rather than only by browsers — the JSP and ASP.NET tracks in Lectures 9–15 both include building the kind of data-access layer such an API ultimately depends on. Web 4.0 (roughly 2020–2024) is often described as intelligent and ubiquitous — AI-assisted interfaces, an Internet of Things (IoT) full of always-connected devices, and an expectation of real-time responsiveness everywhere. Web 5.0 (from roughly 2025 onward) is the most recent, least settled label, generally used to describe an emerging decentralized, “symbiotic” Web in which users are meant to own and control their own data, AI agents act on a user's behalf across services, and blockchain-backed identity systems are explored as an alternative to today's centrally-managed logins — this era is still forming, and much of it remains more aspiration than deployed standard practice as of 2026.

Don't over-trust version labels

Unlike a software version number, “Web 3.0” or “Web 4.0” was never ratified by a single standards body with a precise technical definition — different authors draw the era boundaries slightly differently, and you will find sources that disagree on exact years or on what belongs in which era. Treat these labels the way you would treat “the 1980s” in music history: a useful shorthand for a cluster of trends that mostly happened around the same time, not a precise technical specification. What is NOT in dispute is the underlying HTTP/HTML/URL model from Section 1.2 — that has remained architecturally stable since 1991, which is exactly why a course teaching it today is not already obsolete.

1.6 Programming Languages for the Web: A Snapshot

New web programmers are often surprised there is no single “the” web programming language — the client side and server side of a request typically use entirely different languages, and even within the server side, an enormous range of languages are viable choices. The table below reports the TIOBE Index for August 2026, a widely cited (though not universally endorsed) monthly ranking of overall programming-language popularity based on search-engine and course/vendor query volume, restricted here to languages directly relevant to this course.

Language	TIOBE rank (Aug 2026)	Where it appears in this course
Python	#1 (18.53%)	Not covered in this course (mentioned only for context — DAA and other courses use it/C++ instead).
Java	#4 (8.25%)	Lectures 9–10: JSP runs on the Java platform and inherits its full standard library and tooling.

Language	TIOBE rank (Aug 2026)	Where it appears in this course
C#	#5 (4.09%)	Lectures 13–15: ASP.NET is built on C# and the .NET runtime.
JavaScript	#6 (2.63%)	Client-side interactivity, Lecture 3; the only language that runs directly in every browser.
SQL	#8 (1.88%)	Every one of this course's three CRUD stacks (PHP, JSP, ASP.NET) issues SQL to a relational database.
PHP	#18 (Mar 2026, ~1.2%)	Lectures 5–8: PHP fundamentals, PHP+MySQL, and PHP MVC/Laravel.

Two rankings, two very different pictures of PHP

TIOBE measures search-engine and course-query volume — roughly, "how much are people currently searching about this language" — and by that measure PHP has fallen to around 18th place. But W3Techs measures something entirely different: what server-side language currently-live websites actually run. By that measure, PHP still powers roughly 70% of all websites whose server-side language W3Techs can detect — far more than every other server-side language combined, including ASP.NET's roughly 4% and Java's roughly 6%. Neither number is "wrong"; they are simply answering different questions. This is exactly why this course still devotes four full lectures (5–8) to PHP: raw search-trend popularity and the sheer number of running, revenue-generating systems built in a language are two different things, and as a professional you will very likely maintain far more PHP than its TIOBE ranking alone would suggest.

A note on rankings and up-to-date data

Language-popularity indices such as TIOBE, W3Techs, the Stack Overflow Developer Survey, PYPL, and GitHub's Octoverse report each measure something slightly different — search volume, live website deployments, self-reported developer usage, tutorial queries, or repository activity — and their rankings shift from month to month and year to year. The specific numbers above are current as of August 2026; treat any rankings you read (in this book or elsewhere) as a snapshot of a moving target, and prefer the most recently published edition of whichever index you consult.

1.7 The Three Stacks This Course Will Master

Rather than teach one framework in exhaustive depth, this course deliberately teaches the same problem — build a working, database-backed CRUD web application — three times, in three different server-side technologies, so that you experience firsthand how much of web programming is universal (the request-response cycle, relational database design, CRUD, authentication) and how much is merely syntax.

- PHP + MySQL (Lectures 5–8) — the most widely deployed open-source server-side stack on the Web; you will start with procedural PHP fundamentals, add session/cookie-based state and direct MySQL access, and finish by building the same application with the MVC pattern using the Laravel framework.
- JSP on the Java platform (Lectures 9–12) — Java Server Pages and servlets, running inside a container such as Apache Tomcat, giving you your first exposure to a strongly-typed, compiled server-side language and to the MVC pattern in an enterprise-Java setting.

- ASP.NET + ADO.NET (Lectures 13–16) — Microsoft's server-side web framework, including Web Forms, server controls, validation, state management, and the ADO.NET data-access layer that connects it to a relational database.

Across all three tracks, this book uses a single running case study — a laundry-service management system named Profitina Laundry — so you can directly compare how the identical business problem (managing customers, employees, services, and bills) is solved in each stack. Profitina Laundry is not an invented toy name: it is built and operated under the same Profitina brand you will meet in Section 1.8, one concrete storefront in the wider Profitina product family, which is exactly why it is a fitting shared example for a book that treats Profitina as its running proof that "this is how real systems are actually built." You will meet Profitina Laundry's database schema for the first time in Lecture 10, and implement it in PHP (Lecture 7), JSP (Lecture 10), and ASP.NET (Lecture 14).

1.8 WebPro in Practice: Profitina

> **About this box**
>
> Throughout this book, boxes like this one connect course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, and to Profitina Laundry, the real laundry-service business used as this book's running case study (Section 1.7). These boxes describe WHERE and WHY a concept is used in a live, running system — never the proprietary implementation details, business logic, or source code of Profitina itself, which remain confidential. See the companion document 'Profitina: A Non-Confidential Technical Overview' for the full picture.

Profitina's own public-facing web presence is, underneath its polish, exactly the request-response cycle described in Section 1.3: a visitor's browser resolves profitina.com through DNS, sends an HTTP request to Profitina's web server, and receives back a rendered page describing the product. More importantly, the very problem Profitina exists to solve — how visible a business is when an AI system answers a question about it — depends entirely on the Web's HTTP-and-HTML foundation from Section 1.2: an AI assistant that recommends (or fails to recommend) a business is, in almost every case, ultimately grounded in web pages that some crawler fetched using HTTP and parsed as HTML, exactly like the browser in Figure 1.1. Understanding precisely how a page is requested, served, and rendered — the subject of this entire chapter — is a prerequisite for understanding how a page becomes visible to an AI system at all, which is the layer Profitina operates on. Profitina Laundry sits one layer below that: it is a small, ordinary business whose own visibility, bookings, and billing depend on exactly the CRUD web application you will spend Lectures 5–16 building for it — an everyday reminder that the Web's foundations you are learning in this chapter are not academic exercises, but the same mechanics that keep a real laundry business's doors, and its books, open.

1.9 Chapter Summary and Key Takeaways

- The Internet is the global network; the Web is one application (HTML + HTTP + URLs) that runs on top of it.
- Every ordinary page view is one HTTP request-response cycle: DNS lookup, request, server-side processing, response, and browser rendering.
- HTTP is stateless by design — remembering anything about a visitor across requests requires deliberately added mechanisms such as cookies and sessions (Lecture 6).

- The Web's five commonly cited eras (1.0 read-only, 2.0 read-write, 3.0 API/mobile, 4.0 intelligent/ubiquitous, 5.0 decentralized/symbiotic) are useful shorthand, not precisely standardized version numbers.
- This course teaches the same CRUD web application three times — in PHP/MySQL, JSP, and ASP.NET/ADO.NET — around one shared case study, Profitina Laundry, so you learn what is universal versus what is stack-specific.

"The original idea of the web was that it should be a collaborative space where you can communicate through sharing information." — every technology this course teaches exists, ultimately, to serve that one idea.

1.10 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Lecture 2.

1. Explain, in your own words, why the Internet and the Web are not the same thing. Give one example of an Internet application that is not part of the Web.
2. Walk through the five steps of Figure 1.1 for a concrete case: you type https://profitinalaundry.example.com/bills into your browser's address bar. What happens at each step, and which step(s) involve the database?
3. HTTP is described as "stateless." Give a real example (other than login) of information a web application might need to remember across multiple requests from the same visitor, and briefly suggest how you might make the server "remember" it despite HTTP's statelessness.
4. A GET request is supposed to be "safe" (no server-side side effects). Describe a plausible bug where a developer violates this rule, and explain one concrete problem it could cause (hint: think about what a browser or search-engine crawler might do to a URL automatically, without a human clicking anything).
5. Pick any website or app you use daily. Which of the five Web eras (1.0–5.0) do you think its core functionality most belongs to, and which single feature most influenced your answer?

References

[1] T. Berners-Lee, R. Cailliau. "WorldWideWeb: Proposal for a HyperText Project." CERN, November 1990.

[2] R. Fielding et al. "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content." RFC 7231, Internet Engineering Task Force, June 2014.

[3] TIOBE Software. "TIOBE Index for August 2026." https://www.tiobe.com/tiobe-index/ (accessed August 2026).

[4] W3Techs. "Usage Statistics and Market Share of Web Servers, August 2026." https://w3techs.com/technologies/overview/web_server (accessed August 2026).

[4b] W3Techs. "Usage Statistics and Market Share of Server-Side Programming Languages for Websites, August 2026." https://w3techs.com/technologies/overview/programming_language (accessed August 2026).

[4c] TechRepublic. “TIOBE Index for August 2026: Java Nears C++ as Rust Holds No. 10.” https://www.techrepublic.com/article/news-tiobe-august-2026-java-nears-c-plus-plus/ (accessed August 2026); StatisticsTimes.com, “Top Computer Languages 2026” (PHP ranking, March 2026 data), https://statisticstimes.com/tech/top-computer-languages.php.

[5] T. Berners-Lee. Quoted in: Goodreads, “Quotes by Tim Berners-Lee.” https://www.goodreads.com/author/quotes/428754.Tim_Berners_Lee.

[6] R. Soelaiman, S. Candra, M. M. I. Subakti. “Efficient Approach for Deterministic Data Extrapolation from a Clean Periodic Function with Periodic Components Representation by a System of Linear Equations.” Journal of Theoretical and Applied Information Technology (JATIT), Vol. 97, No. 8, 2019.

CHAPTER 2

HTML: Structuring Content for the Web

Every page Profitina Laundry's customers touch — the homepage, the booking form, the billing table — starts life as plain HTML. This chapter is where that starts.

> **Learning Objectives — after this chapter you will be able to:**
>
> (1) Write a valid HTML5 document with correct doctype, head, and body structure. (2) Choose semantic HTML5 elements (header, nav, main, section, article, aside, footer) over generic divs, and explain why the choice matters to machines as well as people. (3) Mark up text content, images, and tabular data correctly. (4) Build a working HTML form using appropriate input types, labels, and built-in validation attributes. (5) Apply core accessibility practices (alt text, label association, lang, ARIA landmarks) and explain their relationship to how AI crawlers actually read a page.

2.1 Recap: From Lecture 1 to Lecture 2

Lecture 1 built the mental model this whole course runs on: a browser sends an HTTP request, a server answers, and the browser renders whatever HTML comes back. This chapter is the first time you write that HTML yourself. Every example below is a real, working fragment of Profitina Laundry's actual homepage — the same page that will grow a stylesheet in Lecture 3, get form-handling PHP in Lectures 5–7, a JSP rewrite in Lectures 9–10, and an ASP.NET rewrite in Lectures 13–14 (Figure 2.1).

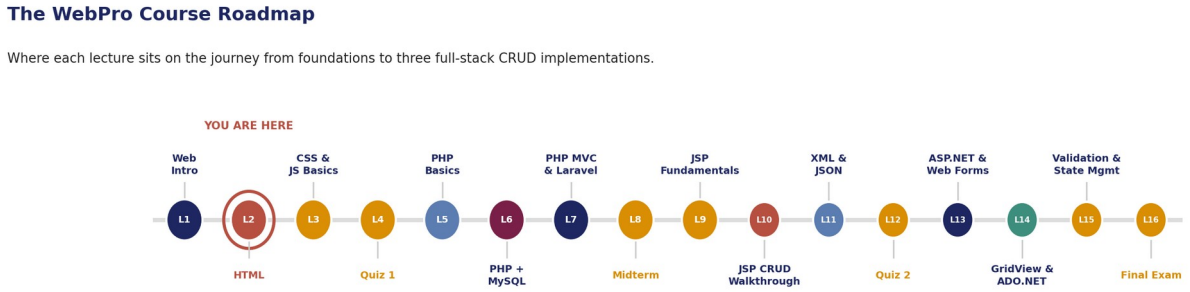

Figure 2.1 — Where this chapter sits in the sixteen-lecture WebPro roadmap. Lecture 3 will add CSS and JavaScript on top of the structure this chapter builds.

2.2 The Anatomy of an HTML5 Document

Every valid HTML5 page — Profitina Laundry's homepage included — nests the same five layers in the same order. Miss one, and the browser is left guessing what you meant, which is exactly the kind of guess that renders differently across browsers, screen readers, and crawlers (Figure 2.2).

Anatomy of an HTML5 Document

Every valid HTML5 page nests the same five layers — skip one and browsers must guess what you meant.

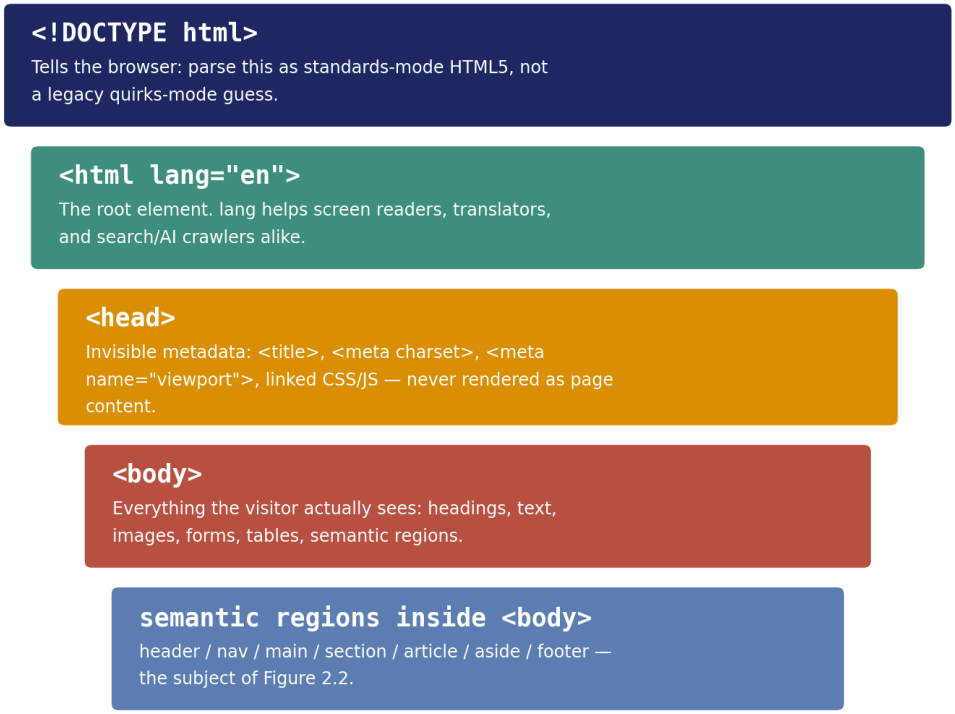

Figure 2.2 — The five nested layers every HTML5 document needs: doctype, root element, head, body, and — inside body — the semantic regions covered in Section 2.4.

> **HTML is a Living Standard, not a fixed version**
>
> "HTML5" is the name most people still use, but there hasn't been an "HTML6" and there never will be. Since 2019 the WHATWG (the industry group of browser makers that includes Google, Apple, Mozilla, and Microsoft) and the W3C have jointly maintained a single specification called the HTML Living Standard — continuously updated rather than frozen into numbered releases. In practice this means: don't write `<!DOCTYPE html5>` (there is no such thing) — the correct, permanent doctype is simply `<!DOCTYPE html>`, and it will stay correct for as long as the Web exists.

Every attribute on the root element matters more than it looks. `<html lang="en">` is one word, but it tells a screen reader which pronunciation rules to use, tells an automatic translator which language it is translating from, and — increasingly relevant to Profitina's own business — tells a search or AI crawler what language the page's content is written in before it even starts parsing the body.

2.3 Text Content: Headings, Paragraphs, Lists, and Links

Text elements are HTML's oldest and most-used building blocks, and they carry meaning beyond their visual size. `<h1>` through `<h6>` form an outline — screen-reader users routinely jump from heading

to heading the way a sighted reader skims a table of contents, so a page should have exactly one `<h1>` (its main subject) and should not skip levels (an `<h3>` should not directly follow an `<h1>`).

- `<p>` — a paragraph of body text; browsers add space above and below it automatically.
- `<ul>` / `<ol>` / `<li>` — unordered and ordered lists; use `<ol>` only when the order itself is meaningful (steps, rankings), `<ul>` otherwise.
- `<a href="...">` — a hyperlink; the visible text should describe the destination ("Book a Pickup", never "click here"), because that text is what a screen reader announces and what a crawler uses to understand what the linked page is about.
- `<strong>` / `<em>` — semantic emphasis (importance / stress), not just bold or italic styling; screen readers can change tone of voice for these, which a purely visual `<b>` or `<i>` will not trigger.

> **Bold text and a bold-looking heading are not the same thing**
>
> Making a `<p>` bold and larger with CSS so it LOOKS like a heading is a common shortcut — and a real accessibility bug. A screen reader's "jump to next heading" command only recognizes actual `<h1>`–`<h6>` elements; a bold paragraph is invisible to it no matter how big the font is. If something is structurally a heading, mark it up as one.

2.4 Semantic HTML5: Structuring a Page Like Profitina Laundry's Homepage

HTML5 added a set of elements whose entire purpose is to name the ROLE a region of the page plays, not just its visual position. Two pages can render as pixel-identical boxes on screen while being read completely differently by anything that isn't a pair of human eyes — a screen reader, a browser's built-in accessibility tree, or a search/AI crawler trying to understand what a page is about (Figure 2.3).

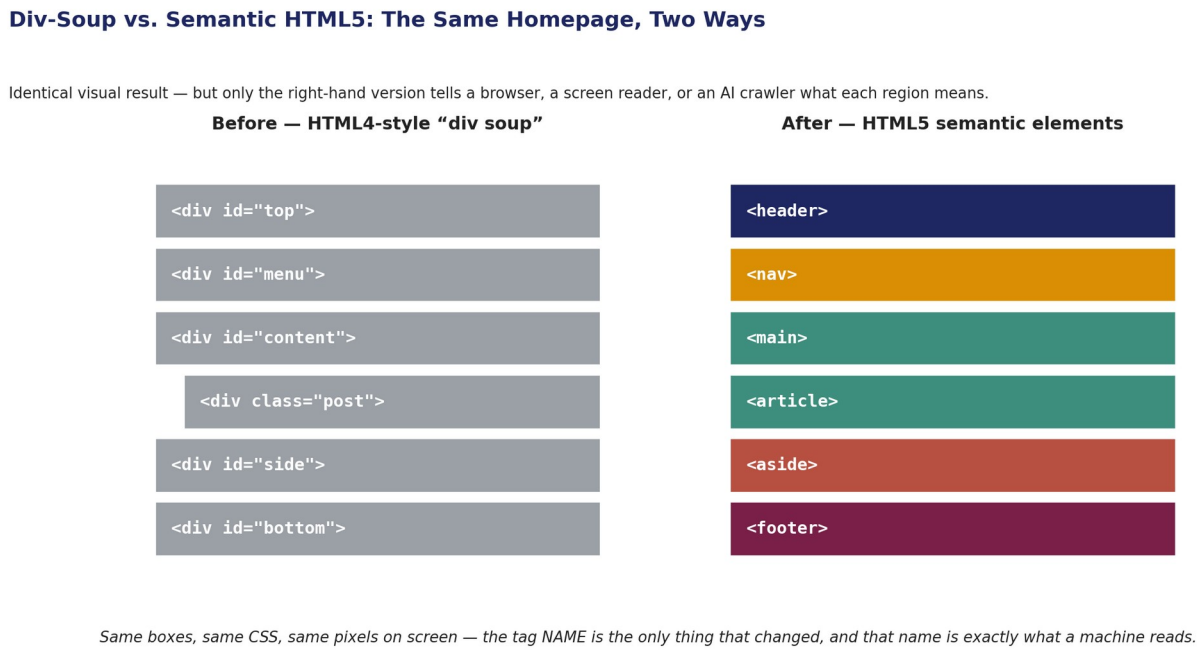

Figure 2.3 — The same visual homepage, marked up two ways. Only the elements on the right announce a role to anything reading the markup instead of looking at it.

2.4.1 Before: Div Soup (the HTML4-era habit to unlearn)

```
<h2>Before: HTML4-style "div soup" (avoid this)</h2>
<div id="top">Profitina Laundry</div>
<div id="menu">
  <div class="menu-item"><a href="#services">Services</a></div>
  <div class="menu-item"><a href="#booking">Book a Pickup</a></div>
</div>
<div id="content">
  <div class="post">
    <div class="post-title">Wash &amp;amp; Fold</div>
    <div class="post-body">Ready in 24 hours.</div>
  </div>
</div>
<div id="side">Customers rate us 4.8 / 5.</div>
<div id="bottom">&copy; 2026 Profitina Laundry</div>
```

2.4.2 After: Six Semantic Landmarks

```
<h2>After: HTML5 semantic elements (what this course teaches)</h2>
<header>Profitina Laundry</header>
<nav>
  <a href="#services">Services</a>
  <a href="#booking">Book a Pickup</a>
</nav>
<main>
  <article>
    <h3>Wash &amp;amp; Fold</h3>
    <p>Ready in 24 hours.</p>
  </article>
</main>
<aside>Customers rate us 4.8 / 5.</aside>
<footer>&copy; 2026 Profitina Laundry</footer>
```

Element	Landmark role it announces	Used exactly once per page?
<header>	Introductory content — logo, site name, primary heading	Usually yes (per page or per article)
<nav>	A block of primarily navigational links	No — a page can have several (main nav, footer nav)
<main>	The one primary, unique content of the page	Yes — exactly one per page, always
<article>	Self-contained content that would make sense syndicated alone	No — Profitina Laundry's homepage has three, one per service
<aside>	Content tangentially related to the main content	No — zero or more
<footer>	Closing content — copyright, contact info, site links	Usually yes (per page or per article)

Notice section 2.4.1's markup and section 2.4.2's markup would produce an IDENTICAL screenshot once the same CSS is applied to both — Lecture 3 could style either one to look like Profitina Laundry's real homepage. The only thing that changed is the tag NAME, and the tag name is precisely what a machine — screen reader, crawler, or otherwise — actually reads.

2.5 Images and Media

An `<img>` element needs a `src` and, just as importantly, an `alt` attribute — the text a screen reader announces in the image's place, and the text a search or AI crawler falls back on when it cannot itself "see" the picture. An empty `alt=""` is the correct choice for a purely decorative image (it tells assistive technology to skip it silently); a missing `alt` attribute entirely is always a mistake, decorative or not.

```
<img src="profitina-laundry-logo.svg"
     alt="Profitina Laundry logo" width="48" height="48">

<figure>
  <img src="wash-fold-basket.jpg" alt="A basket of folded towels">
  <figcaption>Every Wash &amp; Fold order is folded, not just
dried.</figcaption>
</figure>
```

- `<figure>` / `<figcaption>` — pairs an image (or a code sample, chart, or diagram) with a caption that is programmatically associated with it, not just visually nearby.
- `loading="lazy"` on an `<img>` tells the browser to defer downloading images that are off-screen until the visitor scrolls near them — a one-word attribute with a real, measurable effect on how fast a page like a long product catalog first appears usable.
- `srcset` lets one `<img>` offer several resolutions of the same picture so a phone downloads a smaller file than a desktop monitor does, from the exact same markup.

2.6 Tables for Tabular Data

A table is the correct element only for genuinely tabular data — rows and columns of comparable values — never as a page-layout trick (that was a common, and thoroughly retired, HTML4-era hack). Profitina Laundry's billing history is a textbook case for a real table.

```
      <table>
        <caption>Your last four Profitina Laundry bills</caption>
        <thead>
          <tr>
            <th scope="col">Bill #</th>
            <th scope="col">Date</th>
            <th scope="col">Service</th>
            <th scope="col">Amount (IDR)</th>
            <th scope="col">Status</th>
          </tr>
        </thead>
        <tbody>
          <tr>
            <th scope="row">B-1042</th>
            <td>2026-08-18</td>
            <td>Wash &amp; Fold</td>
            <td>85,000</td>
            <td>Paid</td>
          </tr>
```

Element	Purpose
<table>	The table itself
<caption>	A short, programmatically-associated title for the whole table
<thead> / <tbody> / <tfoot>	Groups header rows, body rows, and summary rows so assistive tech and CSS can target each separately
<tr>	One table row
<th scope="col">	A column header cell — `scope` tells a screen reader every cell below it in that column relates to it
<th scope="row">	A row header cell — here, the bill number that every other cell in that row describes
<td>	An ordinary data cell

Why scope="col" and scope="row" matter more than they look

Sighted users infer "this number is the Amount column" by eye, glancing up at the header. A screen reader user navigating cell by cell has no such glance available — without `scope`, "145,000" is just a number with no context. With `scope="col"` on the header row and `scope="row"` on the bill-number cell, a screen reader can announce "Amount (IDR): 145,000, row B-1039" for every single cell automatically. It costs one attribute per header cell and it is the difference between a usable table and an unusable wall of numbers for a real class of Profitina Laundry's own customers.

2.7 Forms: Collecting Data from Users

A `<form>` is how a browser collects and submits user input to a server — the booking request that will, starting in Lecture 5, actually reach a PHP script and write a row into Profitina Laundry's database. Every field needs a `<label>` explicitly tied to its input via `for`/`id`; without that pairing, clicking the label text does nothing and a screen reader cannot announce what the field is for.

```
        <div class="field">
          <label for="customer-email">Email</label>
          <input type="email" id="customer-email" name="customerEmail"
                 required autocomplete="email">
        </div>

        <div class="field">
          <label for="service-type">Service</label>
          <select id="service-type" name="serviceType" required>
            <option value="">Choose a service&hellip;</option>
            <option value="wash-fold">Wash &amp; Fold</option>
            <option value="dry-clean">Dry Cleaning</option>
            <option value="express">Express Same-Day</option>
          </select>
```

Input type	What it's for	Built-in validation you get for free
text	Short free text (a name)	`required`, `minlength`, `maxlength`, `pattern`
email	An email address	Browser rejects text without an @ and a domain

tel	A phone number	No format enforced by default — pair with `pattern`
number	A numeric quantity	`min`, `max`, `step`; browser shows a stepper control
date	A calendar date	Browser renders a native date picker
checkbox	A yes/no toggle	`checked` sets the default; value only submits if checked

Why bother with type="email" if a server must re-check it anyway?

You will learn in Lecture 6 that the server can NEVER trust anything a browser sends — a client-side check like `type="email"` can always be bypassed by a visitor who edits the request directly. So why use it at all? Because it is a courtesy, not a security boundary: it lets 99% of honest users catch a typo instantly, for free, with zero JavaScript and zero server round-trip. Client-side validation improves the experience; server-side validation (Lecture 6 onward) protects the data. A production form needs both, for different reasons.

2.8 Accessibility and HTML's Surprising Role in AI Visibility

Accessibility markup — `alt`, `label`/`for`, `lang`, ARIA landmarks like the `aria-label` and `aria-labelledby` attributes used throughout this chapter's examples — exists first and foremost for human beings using assistive technology. But the same markup increasingly matters for a second audience Profitina was built to think about professionally: the crawlers and language models that decide whether a business is recommended when someone asks an AI assistant a question.

What actually helps an AI crawler read your page (and what doesn't)

It is tempting to assume adding invisible `schema.org`/JSON-LD structured data is the main lever for AI visibility. The evidence is more mixed than that: independent 2025–2026 testing across several AI systems found JSON-LD extraction rates ranging from 0% to about 50% depending on the system, with visible, well-structured HTML content consistently outperforming hidden schema markup for what large language models actually extract — one test even had an AI system pull a fake address out of deliberately invalid JSON-LD, suggesting today's crawlers often treat structured data as ordinary text rather than parsing it strictly. Search engines are a partial exception: Google and Bing do still parse JSON-LD during indexing for Knowledge Graph and rich-result features. The practical, well-supported takeaway for this course: get the semantic HTML and accessible markup in Sections 2.4–2.7 right FIRST — clean headings, real `<nav>`/`<main>`/`<article>` landmarks, descriptive link text, `alt` text, and labeled forms — because that is what both a screen reader AND an AI crawler most reliably understand. Structured data is worth adding later as a supplement, not a substitute.

This is exactly the layer Profitina, the author's AI-visibility product, spends its time analyzing on behalf of real businesses — and it is why this course teaches semantic HTML as a first-class topic in Chapter 2 rather than as an afterthought once a page merely "looks right."

2.9 WebPro in Practice: Profitina

> **About this box**
>
> Throughout this book, boxes like this one connect course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, and to Profitina Laundry, the real laundry business used as this book's running case study. These boxes describe WHERE and WHY a concept is used in a live, running system — never the proprietary implementation details, business logic, or source code of Profitina itself, which remain confidential.

Section 2.8 already told half of this chapter's Profitina story. The other half is Profitina Laundry itself: the semantic homepage in Section 2.4, the booking form in Section 2.7, and the billing table in Section 2.6 are not hypothetical teaching examples invented for this book — they are simplified, teachable versions of the exact page types a real laundry business runs in production, the same page types this course will re-implement against a live database in three different server-side stacks starting in Lecture 5.

2.10 Chapter Summary and Key Takeaways

- Every valid HTML5 document nests the same five layers: doctype, `<html lang>`, `<head>`, `<body>`, and — inside body — semantic regions.
- HTML is the WHATWG/W3C Living Standard — continuously maintained, not released as numbered versions; the correct doctype is simply `<!DOCTYPE html>`.
- Semantic elements (`header`, `nav`, `main`, `article`, `aside`, `footer`) can render pixel-identical to generic `<div>`s while meaning something completely different to a screen reader or crawler.
- Tables need `<caption>`, `<thead>`/`<tbody>`, and `scope` on header cells to be genuinely usable by assistive technology, not just visually organized.
- Forms need a `<label>` explicitly tied to every input, and their `type` attribute buys free client-side validation — which improves experience but never replaces server-side validation (Lecture 6).
- For AI visibility specifically, clean semantic HTML and accessible markup currently outperform hidden structured data as what large language models reliably extract — get Sections 2.4–2.7 right before reaching for schema.org.

"Content precedes design. Design in advance of content is not design, it's decoration."

— Jeffrey Zeldman

HTML is where content precedes everything else — CSS (Lecture 3) only ever decorates what Lecture 2 structures first.

2.11 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Lecture 3.

1. Take the "div soup" excerpt in Section 2.4.1 and rewrite it, element by element, using the six semantic landmarks from Section 2.4.2's table. Which HTML4-era div doesn't map cleanly onto any single HTML5 landmark, and what does that tell you about the limits of the semantic vocabulary?
2. Profitina Laundry's booking form (Section 2.7) has no client-side check preventing a `pickup-date` in the past. Propose an HTML-only attribute-based fix (no JavaScript), and explain one case where even that fix would not be enough.
3. Explain, in your own words, why `alt=""` (empty) and a missing `alt` attribute entirely are not the same thing, and give one concrete image on Profitina Laundry's homepage where each would be the correct choice.
4. Section 2.8 argues that semantic HTML currently matters more for AI visibility than hidden JSON-LD structured data. Summarize the evidence given for this claim in two sentences, and identify one caveat the chapter itself acknowledges.
5. Explain why a table is the correct element for Profitina Laundry's billing history (Section 2.6) but would be the WRONG choice for laying out the three-column services section (Section 2.4) — even though both are visually "grids" of boxes.

Appendix 2.A — Validation Log for Chapter 2 Source Code

Both HTML files provided with this chapter (`profitina-laundry-home.html` and `semantic-vs-divsoup.html`) were parsed with `html5lib`'s strict HTML5 parser — the same parsing algorithm the HTML Living Standard specifies browsers must implement — and confirmed free of structural errors before being excerpted into this chapter.

```
$ python3 -c "import html5lib; ..."
profitina-laundry-home.html : PARSED OK, no strict errors
semantic-vs-divsoup.html    : PARSED OK, no strict errors
```

References

[1] WHATWG. “HTML Living Standard.” https://html.spec.whatwg.org/ (last updated August 26, 2026; accessed August 2026).

[2] WHATWG/W3C. “Memorandum of Understanding between W3C and WHATWG,” 2019 — the agreement establishing the HTML Living Standard as the single, jointly-maintained specification, ending the separate “HTML5” versioned track.

[3] MDN Web Docs. “HTML: HyperText Markup Language” and “ARIA” reference sections. https://developer.mozilla.org/en-US/docs/Web/HTML (accessed August 2026).

[4] Ryall, P. “Structured Data & AI Crawlers: The Schema Hype, Debunked.” 2026. https://patrickryall.com/articles/structured-data-ai-crawlers (accessed August 2026) — source for the JSON-LD extraction-rate findings discussed in Section 2.8.

[5] Zeldman, J. Quoted in: Goodreads, “Quotes by Jeffrey Zeldman.” https://www.goodreads.com/quotes/9357188 (accessed August 2026).

[6] World Wide Web Consortium (W3C). Web Content Accessibility Guidelines (WCAG), section on table headers and form labeling. https://www.w3.org/WAI/ (accessed August 2026).

CHAPTER 3

CSS & JavaScript Basics

Chapter 2 gave Profitina Laundry's homepage structure and meaning. This chapter gives it a look and a pulse: CSS decides how every element is boxed, spaced, and arranged; JavaScript makes it respond to what a visitor actually does.

Learning Objectives — after this chapter you will be able to:

(1) Attach CSS to a page three ways, and explain why an external stylesheet is almost always the right one. (2) Reason about the CSS box model, selectors, and the cascade well enough to predict which rule wins. (3) Lay out a real page region with Flexbox and with CSS Grid, and know which one to reach for. (4) Use native CSS nesting and a size Container Query, and describe one caveat each brings. (5) Declare variables with let/const and write arrow functions. (6) Select DOM elements, attach event listeners, and fetch JSON data with the Fetch API using async/await.

3.1 Recap: From Lecture 2 to Lecture 3

Lecture 2 built Profitina Laundry's homepage as pure, unstyled HTML — correct structure, correct semantics, zero visual design. Open that file in a browser today and it renders as a plain, top-to-bottom stack of black text on white: every heading, list, and table is there, but nothing is arranged, colored, or interactive yet. This chapter changes exactly that, without touching the HTML's structure at all — every example below styles and scripts the SAME markup Chapter 2 already built, through two new files this chapter adds: `styles.css` and `script.js` (Figure 3.1).

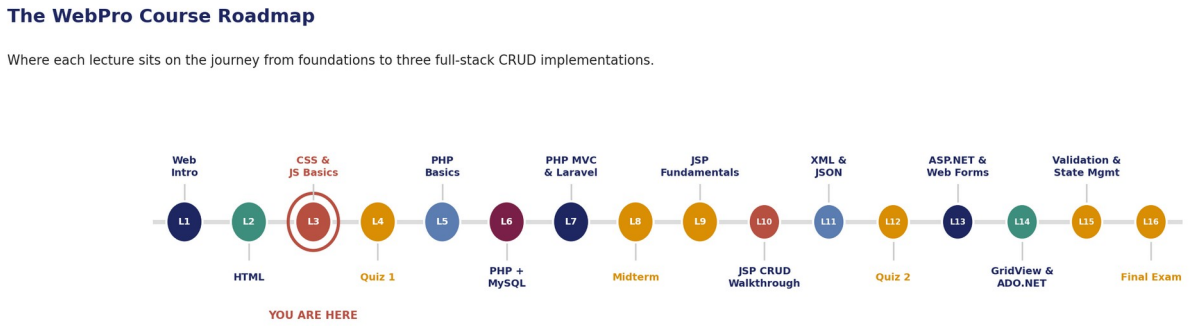

Figure 3.1 — Where this chapter sits in the sixteen-lecture WebPro roadmap. Lecture 4 is Quiz 1, covering everything through this chapter.

3.2 Three Ways to Add CSS to a Page

A browser accepts CSS from three places, and real projects should only ever use one of them for anything beyond a quick test.

- Inline — a `style="..."` attribute directly on one HTML element. Highest priority in the cascade, but unmaintainable at scale: styling a hundred buttons this way means editing a hundred attributes to change one color.

- Internal — a `<style>` block inside the page's own `<head>`. Fine for a single self-contained demo page, but the styles cannot be reused by any other page on the site.
- External — a separate `.css` file linked with `<link rel="stylesheet" href="styles.css">`, exactly what Chapter 2's homepage already had waiting in its `<head>`. One file, cached once by the browser, reused by every page on Profitina Laundry's site.

> **styles.css, linked and ready since Chapter 2**
>
> Chapter 2's `<head>` already contained `<link rel="stylesheet" href="styles.css">` — the file simply didn't exist yet. This chapter writes it, and the browser picks up every rule in it the moment the file exists, with no change to the HTML at all.

3.3 Selectors, Specificity, and the Cascade

A CSS rule is a selector (what to style) paired with a declaration block (how to style it). The three selector types used constantly in this chapter's stylesheet are the element selector (`header`, matching every `<header>`), the class selector (`.field`, matching every element with `class="field"`), and the ID selector (`#services`, matching the one element with `id="services"`).

When two rules could both apply to the same element, the "Cascading" in Cascading Style Sheets decides which one wins, using — in order — origin (a browser default loses to an author stylesheet), specificity (an ID beats a class, a class beats a bare element), and finally source order (the rule written later in the file wins a true tie). A full specificity calculus exists, but the practical rule this chapter leans on is simpler: prefer classes for almost everything, reach for an ID only for genuinely page-unique regions like `#services`, and avoid inline styles entirely so the cascade stays predictable.

> **"!important" is not a specificity shortcut**
>
> Appending `!important` to a declaration overrides the entire cascade calculation above — but every `!important` makes the NEXT override even harder, since only another `!important` (with a later source order) can beat it. `styles.css` uses zero `!important` declarations; if a rule genuinely isn't winning, the fix is almost always a more specific — or better-ordered — selector, not a bigger hammer.

3.4 The CSS Box Model

Every element a browser renders is, underneath, four nested rectangles: content, padding, border, and margin, from the inside out. Getting this model wrong is the single most common source of "why is there a gap I didn't ask for" bugs in a new stylesheet (Figure 3.2).

The CSS Box Model

Every element is four nested boxes. box-sizing: border-box (Section 3.4) keeps padding and border INSIDE the declared width.

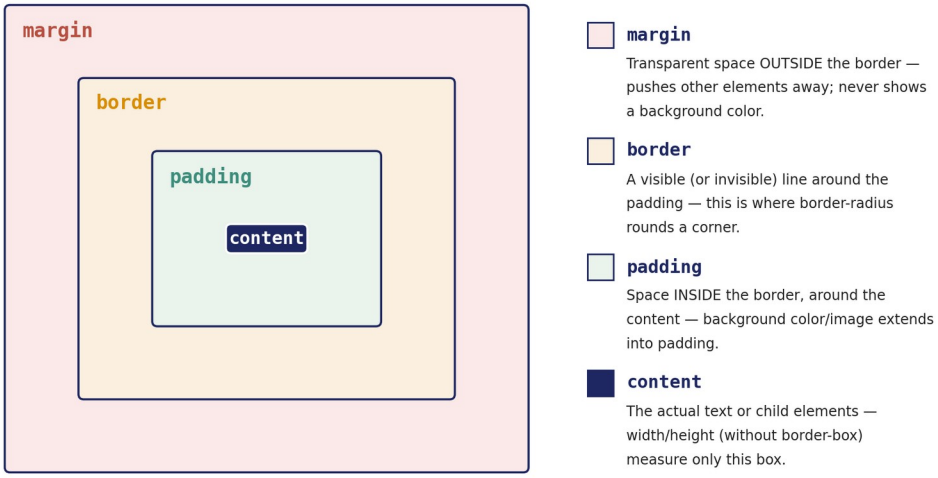

Figure 3.2 — The four boxes every element has, from the inside out. Margin never paints a background color; padding does.

```
/* ---- 3.4 The box model: box-sizing changes what "width" measures ----
   Without this rule, adding padding/border to an element with a fixed
   width makes the element LARGER than that width (content-box, the
   default). border-box instead keeps padding+border INSIDE the
   declared width — the behavior almost every real stylesheet wants. */
*, *::before, *::after {
  box-sizing: border-box;
}
```

Without `box-sizing: border-box`, the browser's default (`content-box`) means a `width: 300px` box that also has `padding: 20px` and a `2px` border actually occupies `344px` on screen — the padding and border are ADDED on top of the declared width. `border-box`, applied once to every element via the universal selector (`*`) at the top of `styles.css`, makes `width` describe the box's total on-screen size instead, which is what almost every layout intuition expects.

3.5 Layout with Flexbox

Flexbox turns an element's direct children into a row (or column) of flexible items that a handful of properties can align, space, and reorder — exactly the job `<header>`'s three children (logo, tagline, and nav) need.

```
/* ---- 3.5 Flexbox: the header row ----
   display:flex turns <header>'s direct children into flex items laid
   out in a row by default. align-items:center vertically centers the
   logo, tagline, and nav despite their different heights; gap puts
   space BETWEEN items without needing margin on each one. */
header {
  display: flex;
  align-items: center;
  gap: 1rem;
  padding: 0.75rem 1.25rem;
  background: var(--navy);

  /* Native CSS nesting (3.7): a selector for a descendant, written
     directly inside the parent rule instead of as a separate
     top-level rule further down the file. */
  & .tagline {
    color: white;
    font-weight: bold;
    font-size: 1.1rem;
    margin: 0;
    margin-right: auto; /* pushes everything after it to the right */
  }
}
```

Property	Set on...	What it does here
display: flex	the parent (header)	turns direct children into flex items, laid out in a row by default
align-items: center	the parent	vertically centers items of different heights along the row
gap: 1rem	the parent	puts space BETWEEN items only — no margin needed on each child
margin-right: auto	one child (.tagline)	consumes all remaining space, pushing everything after it to the right edge

That last row is worth dwelling on: `margin: auto` on a single flex item is how Profitina Laundry's nav ends up flush against the header's right edge without a single pixel of manual positioning — the flex container simply gives the auto-margin every spare pixel it has.

3.6 Layout with CSS Grid and a Container Query

Grid is Flexbox's counterpart for two-dimensional layout — rows AND columns at once — which is exactly what the three service cards in Section "Our Services" need: a grid that shows one column on a narrow screen and three columns once there is room.

```css
/* ---- 3.6 CSS Grid + Container Queries: the services section ----
   container-type:inline-size turns #services into a "query container"
   — its OWN width, not the browser viewport's width, is what the
   @container rule below reacts to. That matters because Section 3.6's
   whole point is: a component should adapt to the space IT has been
   given, not just to the size of the whole screen. */
#services {
  container-type: inline-size;
  container-name: services;
  padding: 1.5rem 0;
}

.card-grid {
  display: grid;
  grid-template-columns: 1fr;   /* one column until the container says otherwise
*/
  gap: 1.25rem;
}

.card-grid article {
  background: var(--bg-light);
  border: 1px solid var(--border);
  border-radius: 0.5rem;
  padding: 1.25rem;

  & h3 { color: var(--teal); margin-top: 0; }
}

/* Fires based on #services's own width (min 640px of container space),
   which is NOT the same threshold as the page-wide nav media query
   above — a container can cross this breakpoint even on a phone, if
   it's placed inside a wide enough layout region. */
@container services (min-width: 640px) {
  .card-grid { grid-template-columns: repeat(3, 1fr); }
}
```

Why the breakpoint here is a Container Query, not a media query

A `@media` query asks "how wide is the whole browser window?" — a `@container` query instead asks "how wide is THIS component's own box?", set up by marking `#services` as a query container with `container-type: inline-size`. The difference matters the moment a component gets reused somewhere narrower than the full page — a sidebar widget, for instance — where a page-wide media query would still report the whole window as "wide" even though the component itself only has a sliver of it.

Container Queries are current, mainstream CSS in 2026

Size Container Queries shipped across Chrome, Firefox, and Safari by early-to-mid 2023 and are solidly Baseline "Widely Available" today — safe to use in production without a fallback. Flexbox and Grid themselves remain formally W3C Candidate Recommendation Drafts rather than final Recommendations, but both have had universal, stable browser support for years; "CR-level and de facto stable" describes most of the CSS a working front-end developer uses daily.

3.7 Modern CSS: Native Nesting

Every nested rule inside `styles.css` — a `&` selector written inside its parent rule instead of as its own separate, repeated top-level selector — is native browser CSS, not a Sass or LESS preprocessor step. Chrome, Firefox, and Safari all shipped support within about three and a half months of each other in late 2023, and as of mid-2026 the feature has just crossed the industry's "Baseline: Widely Available" threshold (roughly 92–93% of browsers in use).

```
header nav ul {
  display: flex;
  gap: 1.25rem;
  list-style: none;
  margin: 0;
  padding: 0;

  /* Nested rule: targets the <a> inside every <li> inside this <ul>,
     without writing "header nav ul li a" as its own top-level line. */
  & li a {
    color: white;
    text-decoration: none;
    font-size: 0.95rem;

    /* Nesting a pseudo-class one level deeper still. Nested rules like
       this are implicitly wrapped in :is(...) by the browser, which is
       why a very specific outer selector can occasionally change how
       specificity is calculated versus writing the same rule flat —
       worth remembering the first time a nested override "loses" to a
       rule you expected it to beat (Section 3.7). */
    &:hover, &:focus {
      color: var(--gold);
      text-decoration: underline;
    }
  }
}
```

A nested selector is implicitly wrapped in :is(...)

The browser treats `header nav ul { & li a { … } }` as equivalent to `:is(header nav ul) li a { … }`, not literally `header nav ul li a { … }` — usually invisible, but it can change a specificity calculation the moment a nested rule and a flat rule targeting the same element compete, which is the most common surprise reported when teams first adopt native nesting.

3.8 JavaScript Basics: Variables and Functions

This chapter's `script.js` uses `const` for every value that is never reassigned and `let` for the handful that are — and never `var`. Both `const` and `let` are block-scoped (visible only inside the `{ }` where they're declared), unlike `var`, which is scoped to the nearest enclosing function and can leak in confusing ways across `if`/`for` blocks. `const`/`let` and arrow functions (`() => { … }`) both arrived together in ECMAScript 2015 (ES6, ratified June 2015) and are treated as the settled, default way to write modern JavaScript.

```
const navToggle = document.querySelector(".nav-toggle");
const primaryNav = document.getElementById("primary-nav");

// ---- 3.9 The DOM: reading and reacting to the page ----
// document.querySelector / getElementById turn a piece of markup into
// a JavaScript object this script can inspect and change. An event
// listener is how that object is told "run this function whenever X
// happens" — here, "toggle the mobile menu whenever the button is
// clicked."
if (navToggle && primaryNav) {
  // Arrow function: a shorter function syntax that also does not
  // rebind `this`, which matters once event handlers get more complex
  // than this one-line example.
  navToggle.addEventListener("click", () => {
    const isOpen = primaryNav.classList.toggle("nav-open");
    navToggle.setAttribute("aria-expanded", String(isOpen));
  });
}
```

The arrow function passed to `addEventListener` above is a callback: a function handed to another function (here, the browser's event system) to be run later, whenever the described event — a click on `.nav-toggle` — actually happens. `classList.toggle("nav-open")` both adds the class if it's missing and removes it if it's present, and conveniently returns `true`/`false` for whichever it just did, which is exactly the value the accessible `aria-expanded` attribute needs.

3.9 The DOM: JavaScript's Model of the Page

JavaScript running in a browser never reads the HTML text file directly — the browser parses that file once into the Document Object Model (DOM), a tree of node objects, and every DOM API (`document.querySelector`, `getElementById`, `classList`, `addEventListener`) reads or changes THAT tree, which the browser then re-renders (Figure 3.3).

The DOM: Profitina Laundry's Homepage as a Tree

JavaScript never sees the HTML text file — it sees this tree of node objects, which document.querySelector() and friends search and change.

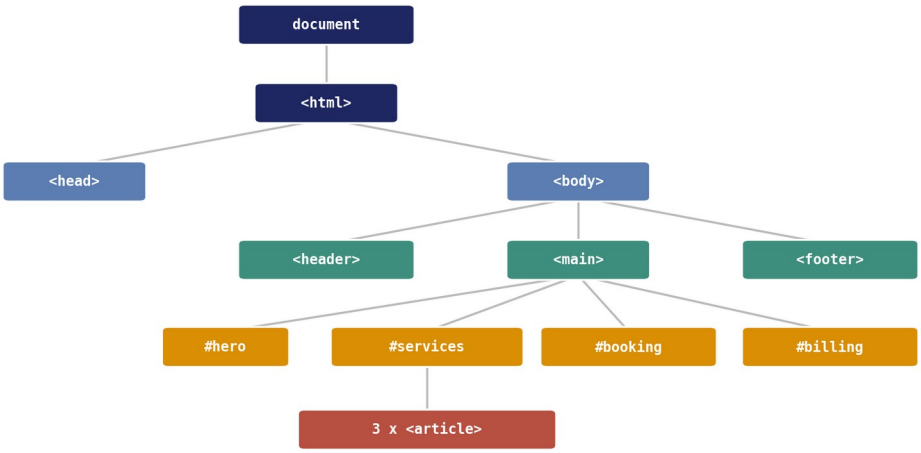

Figure 3.3 — A simplified DOM tree for Profitina Laundry's homepage. document.querySelector("#services") walks exactly this structure to find one node.

```
const notesField = document.getElementById("notes");
const notesRemaining = document.getElementById("notes-remaining");

if (notesField && notesRemaining) {
  const maxLength = Number(notesField.getAttribute("maxlength")) || 120;

  const updateRemaining = () => {
    const remaining = maxLength - notesField.value.length;
    notesRemaining.textContent = String(remaining);
  };

  notesField.addEventListener("input", updateRemaining);
  updateRemaining(); // run once immediately, so the count is correct on page
load
}
```

This second, independent example is deliberately simple: one `input` event listener recalculates a remaining-character count every time the visitor types in the notes `<textarea>`, and `updateRemaining()` is also called once immediately on page load so the count is correct even before anyone has typed anything — a small habit ("run the handler once up front, then let events keep it updated") that shows up constantly in real front-end code.

3.10 The Fetch API: Client-Side Data Without a Page Reload

Every example so far only touched elements already sitting in the HTML. `fetch()` is how a page asks a server (or, for now, a static file) for NEW data without navigating to a new page at all — the modern, Promise-based replacement for the older `XMLHttpRequest`, and the foundation every later WebPro lecture's dynamic, database-backed page builds on.

```
// ---- 3.10 The Fetch API: client-side data without a page reload ----
// fetch() replaces the older XMLHttpRequest for making HTTP requests
// from the browser. It returns a Promise, which is why this uses
// async/await instead of the .then()-chaining style older code uses.
// Lectures 5–7, 9–10, and 13–14 will point this same request at a real
// PHP/JSP/ASP.NET endpoint instead of a static JSON file — the DOM
// code below does not need to change at all when that happens.
const serviceSelect = document.getElementById("service-type");
const weightInput = document.getElementById("weight-kg");
const priceEstimate = document.getElementById("price-estimate");

async function loadServicePrices() {
  const response = await fetch("services.json");
  if (!response.ok) {
    throw new Error(`Could not load services.json: HTTP ${response.status}`);
  }
  // response.json() also returns a Promise, resolving to a parsed object.
  return response.json();
}
```

`async`/`await` is syntax that makes Promise-based code read top-to-bottom like ordinary synchronous code, instead of nested `.then()` callbacks: `await fetch(...)` pauses this function (without freezing the browser) until the network response arrives, and `await response.json()` similarly pauses until the response body has finished being parsed as JSON.

```
function updateEstimate(prices) {
  const serviceKey = serviceSelect.value;
  const weight = Number(weightInput.value);

  if (!serviceKey || !prices[serviceKey] || !(weight > 0)) {
    priceEstimate.textContent = "Choose a service and weight for an estimate.";
    return;
  }

  const service = prices[serviceKey];
  const total = service.pricePerKg * weight;
  const perKg = service.pricePerKg.toLocaleString("id-ID");
  // Template literal: embeds expressions directly in a string with
  // ${...}, instead of concatenating pieces with +.
  priceEstimate.textContent =
    `Estimated price: Rp ${total.toLocaleString("id-ID")} ` +
    `(${service.label}, ${weight} kg at Rp ${perKg}/kg)`;
```

Piece	What it does
fetch("services.json")	requests the file; resolves once the response's headers have arrived (not necessarily the whole body)
response.ok / response.status	true only for a 2xx HTTP status — a 404 does NOT reject the fetch Promise by itself, which is why `!response.ok` is checked explicitly
response.json()	reads and parses the response body as JSON; itself returns a Promise
Template literal `` `text ${expr}` ``	embeds an expression's value directly in a string, replacing string concatenation with `+`

Lectures 5–7, 9–10, and 13–14 will point this exact same `fetch("...")` call at a real PHP, JSP, or ASP.NET endpoint instead of a static `services.json` file, backed by Profitina Laundry's real database — and none of the DOM code in Section 3.9 will need to change at all when that happens, because from JavaScript's point of view a parsed JSON object looks identical whether it came from a static file or a live server.

3.11 WebPro in Practice: Profitina

> **About this box**
>
> Throughout this book, boxes like this one connect course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, and to Profitina Laundry, the real laundry business used as this book's running case study. These boxes describe WHERE and WHY a concept is used in a live, running system — never the proprietary implementation details, business logic, or source code of Profitina itself, which remain confidential.

Profitina's own marketing site, profitina.com, is a WordPress theme — meaning every visual rule this chapter teaches (the box model, Flexbox, Grid, native nesting, container queries) is exactly what that theme's stylesheet is built from, at production scale, in both the Indonesian and English language versions the Polylang plugin serves. And the front end of app.profitina.com, Profitina's actual product, is a browser page like any other: it, too, runs on `fetch()` calls returning JSON — the same Fetch-API pattern Section 3.10 just taught, just aimed at Profitina's own FastAPI backend instead of a static `services.json` file. The self-hosted Qwen2.5 model that powers Profitina's AI analysis is invisible to this layer entirely — the browser only ever sees JSON coming back from an endpoint, exactly like the price-estimate example above, whether the server computed that JSON from a database row or from a locally-run language model.

3.12 Chapter Summary and Key Takeaways

- Use an external stylesheet for anything beyond a one-off demo; `<link rel="stylesheet">` is how Chapter 2's homepage was already wired to receive it.
- `box-sizing: border-box` makes width/height describe an element's total on-screen size, padding and border included — set it once, on every element, at the top of the stylesheet.
- Flexbox arranges one dimension of children (a row or column) with alignment and spacing properties; Grid arranges two dimensions (rows AND columns) at once.
- A Container Query reacts to a component's OWN box width, not the browser window's — different from, and often more useful than, a page-wide media query.
- Native CSS nesting (`&`) is real, current browser CSS as of 2026, not a preprocessor feature — but nested selectors are implicitly wrapped in `:is(...)`, which can shift specificity.
- `const`/`let` and arrow functions are the settled modern JavaScript defaults (ES2015); the DOM is the live tree JavaScript actually reads and writes, never the original HTML text.
- `fetch()` with `async`/`await` is the current standard way to request data from JavaScript — and the same pattern this chapter used against a static JSON file is exactly what a real PHP/JSP/ASP.NET-backed page will use starting in Lecture 5.

> ***"In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders."***
>
> **— Douglas Crockford, JavaScript: The Good Parts (2008)**
>
> *This chapter teaches the elegant parts — let/const, arrow functions, the DOM, and fetch() — the same parts a well-written 2026 codebase actually uses.*

3.13 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Lecture 4 (Quiz 1).

1. Explain, using the box model, why a `<div>` with `width: 200px`, `padding: 10px`, and a `1px` border occupies 222px on screen under `content-box` but exactly 200px under `border-box`.
2. Profitina Laundry's `.card-grid` shows one column below 640px of CONTAINER width and three columns above it. Describe a real layout situation where that would behave differently from a page-wide `@media (min-width: 640px)` rule, and explain why.
3. Rewrite the nested `header nav ul { & li a { &:hover { … } } }` rule from Section 3.7 as three separate, flat, non-nested rules. Which version is easier to read, and which is easier to search a large file for?
4. Section 3.9's character-count example calls `updateRemaining()` once immediately, in addition to attaching it as an event listener. What would the visitor see on page load if that immediate call were removed, and why?
5. `updateEstimate()` in Section 3.10 checks `!response.ok` explicitly rather than relying on `fetch()` to reject on a 404. Explain, in your own words, why a 404 response does not by itself cause a fetch Promise to reject.

Appendix 3.A — Validation Log for Chapter 3 Source Code

This chapter's three new source files were each checked with the parser or tool that matches its language, the same discipline Chapter 2 established with `html5lib` for HTML: `html5lib`'s strict HTML5 parser for the updated `profitina-laundry-home.html`, `tinycss2` (a standards-based CSS tokenizer/parser) for `styles.css`, and Node.js's own `node --check` syntax verifier for `script.js`. All three passed with zero errors before any excerpt in this chapter was taken from them. The interactive behavior itself — the mobile nav toggle, the live character counter, and the Fetch-API price estimator — was additionally exercised end-to-end in a real, running browser (Chromium via Playwright) at both a wide and a narrow viewport before being described in this chapter, confirming the Container Query breakpoint, the computed price, and the character count all behave exactly as Sections 3.6, 3.9, and 3.10 describe.

```
profitina-laundry-home.html : PARSED OK (html5lib, strict), no errors
styles.css                  : PARSED OK (tinycss2), 29 rules, 0 errors
script.js                   : PARSED OK (node --check), no syntax errors

Browser check (Chromium/Playwright), 6kg Dry Cleaning selected:
  #price-estimate -> "Estimated price: Rp 210.000 (Dry Cleaning, 6 kg ...)"
  #notes-remaining -> live-updates on every keystroke, as typed
  .card-grid columns  -> 1 column under 640px container width, 3 above it
```

References

[1] MDN Web Docs. “CSS Box Model.” https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_box_sizing (accessed August 2026).

[2] MDN Web Docs. “Basic Concepts of Flexbox” and “CSS Grid Layout.” https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_flexible_box_layout, https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_grid_layout (accessed August 2026).

[3] MDN Web Docs. “CSS nesting.” https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_nesting (accessed August 2026) — source for native CSS nesting's 2023 browser rollout and the `:is(...)` specificity behavior.

[4] W3C. “CSS Containment Module Level 3” (Container Queries) and web-features Baseline tracker. https://developer.mozilla.org/en-US/docs/Web/CSS/Guides/Containment/Container_queries (accessed August 2026).

[5] W3C. “CSS Flexible Box Layout Module Level 1” (css-flexbox-1) and “CSS Grid Layout Module Level 2” (css-grid-2). https://www.w3.org/TR/css-flexbox-1/, https://www.w3.org/TR/css-grid-2/ (accessed August 2026) — Candidate Recommendation Draft status as of the 2025 republications cited in Section 3.6.

[6] MDN Web Docs. “Using the Fetch API.” https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch (accessed August 2026).

[7] Crockford, D. JavaScript: The Good Parts. O'Reilly Media, 2008 — source for the Preface quotation in Section 3.12.

[8] ECMA International. ECMA-262, ECMAScript 2015 Language Specification (6th Edition) — source for `let`/`const` and arrow function syntax, ratified June 2015.

CHAPTER 4 · EXERCISE CHAPTER

Quiz 1: A Small, Real Take-Home Website Project

No new material here — a small, integrated take-home project drawing together everything Lectures 1 through 3 taught, exactly the way the real Quiz 1 assessment does.

Learning Objectives — after working through this chapter you will be able to:

(1) Explain, in your own words, the client-server model and the HTTP request/response cycle for a real page load. (2) Design a small multi-section page using correct, semantic HTML5 structure. (3) Style that page with an external stylesheet, applying the box model correctly and laying out at least one region with Flexbox or Grid. (4) Add at least one genuine JavaScript interactive feature using DOM selection and event listeners. (5) Fetch and display JSON data client-side with the Fetch API. (6) Package a small project as a report plus a GitHub repository, the same deliverable shape every WebPro assessment uses.

4.1 Recap: Lectures 1–3 in Brief

Lecture 1 laid the conceptual groundwork: the distinction between the Internet (the physical and logical network) and the Web (one application running on top of it), the client-server model, the HTTP request/response cycle, and the Web's own history from static Web 1.0 pages through today. Lecture 2 turned that theory into a real, correctly-structured page: Profitina Laundry's homepage, built from semantic HTML5 — headings in a real hierarchy, `<header>`/`<main>`/`<footer>` landmarks, and markup an assistive technology can actually navigate. Lecture 3 gave that same page a look and a pulse without touching its structure at all: an external stylesheet applying the box model, the cascade, Flexbox and Grid, native CSS nesting and a Container Query — plus a first `script.js`, using modern `let`/`const`, arrow functions, DOM selection, event listeners, and the Fetch API against a static JSON file (Figure 4.1).

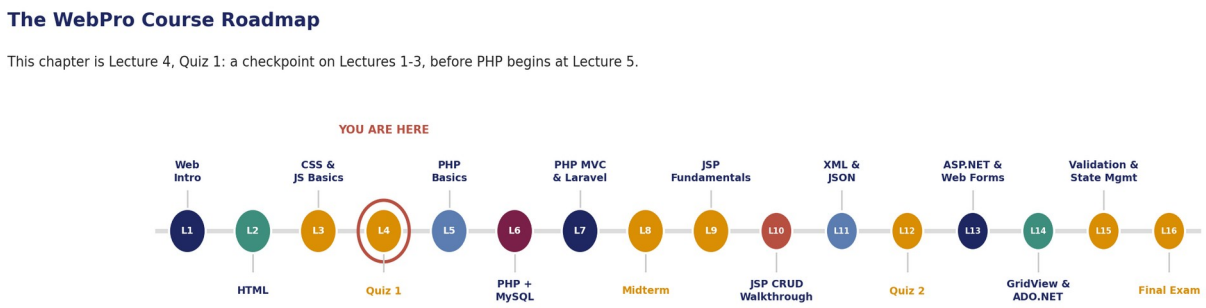

Figure 4.1 — This chapter sits at Lecture 4 in the sixteen-lecture WebPro roadmap: a checkpoint, not a new topic, Quiz 1 before Lecture 5 begins PHP.

4.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a Bab Latihan: a practice chapter, not a new-content chapter. Unlike a course that tests Lectures 1-3 with discrete, independent problems, WebPro's real Quiz 1 — like every WebPro assessment — is a single small take-home PROJECT: one working mini-site, submitted as a report plus a GitHub repository, not a set of numbered short-answer questions. Section 4.3 below walks through that project's four deliverables (each traceable to a specific lecture, see Figure 4.2), each with guidance toward good practice rather than a full worked solution or reference source code. Like Chapters 8, 12, and 16 to come, this chapter ships no code/ subfolder — the project is yours to design and build.

Where Each Mini-Project Deliverable Comes From

Quiz 1 is one small take-home website project -- every deliverable in Section 4.3 traces back to Lectures 1-3.

#	Mini-Project Deliverable	Traces back to
1	Client-Server & HTTP Explanation	L1 -- Internet vs. Web, the HTTP request/response cycle, client/server roles
2	Semantic HTML Structure	L2 -- semantic HTML5 tags, heading hierarchy, accessible markup
3	CSS Layout & Responsiveness	L3 -- box model, cascade, Flexbox/Grid, a Container Query breakpoint
4	JavaScript Interactivity & Fetch	L3 -- DOM selection, event listeners, let/const, fetch() with async/await

Figure 4.2 — Every deliverable in Section 4.3 traces back to Lecture 1, 2, or 3.

Before you start

Sketch your mini-project's page structure and its report outline BEFORE writing any HTML — the same discipline Section 4.3.2 asks you to demonstrate. Quiz 1 rewards a small project that clearly does a few things correctly over a large project that does many things carelessly.

4.3 The Quiz 1 Mini-Project

Build a small, multi-page (or multi-section single-page) static website on a topic of your choosing — it can extend Profitina Laundry with a new page, or be an entirely different small site, as long as it demonstrates every deliverable below.

4.3.1 Choosing Your Mini-Project Topic

Pick a topic simple enough to finish well in the time available, but real enough to need at least three distinct content sections (for example: a hero/introduction section, a services or features section, and a contact or booking section) — the same three-or-more-section shape Profitina Laundry's own homepage already has.

> **Hint**
>
> A topic you can describe in one sentence, with a clear idea of who its three-plus sections are for, is usually the right size. If you find yourself planning more than about five sections, you are probably overscoping a Quiz 1 project.

4.3.2 Required Deliverables & Report

Submit your project as a GitHub repository (source files) plus a short written report. The report should briefly cover: your topic and its target audience, a one-paragraph explanation of the client-server model and HTTP request/response cycle IN YOUR OWN WORDS as it applies to your specific project, and a short account of which HTML/CSS/JavaScript techniques you used and why.

> **Why the client-server explanation belongs in the report, not just the code**
>
> Lecture 1's material is conceptual — there is no HTML tag or CSS rule that "proves" you understood the client-server model. The report is where that understanding becomes visible to a grader, in your own words, tied to your own project.

4.3.3 Semantic HTML Structure Checklist

Structure your page using real HTML5 landmark elements — `<header>`, `<nav>`, `<main>`, appropriate section elements, and `<footer>` — with a heading hierarchy that never skips a level (an `<h1>` followed directly by an `<h3>`, with no `<h2>` in between, is the single most common semantic mistake).

> **Hint**
>
> Open your finished page and mentally delete every CSS rule — read only the raw HTML top to bottom. If the content still makes logical sense in that order, your structure is doing its job independent of how it eventually looks.

4.3.4 CSS Layout & Responsiveness Checklist

Style your page with an external stylesheet (never inline styles, and no more than a token use of an internal `<style>` block). Apply `box-sizing: border-box` globally, and lay out at least one region — a header, a card grid, or similar — with Flexbox or CSS Grid so it visibly rearranges at a narrower width.

> **Hint**
>
> Decide Flexbox vs. Grid by asking whether the region you're laying out needs ONE dimension of control (a row of nav items — Flexbox) or TWO dimensions at once (a grid of cards that should wrap responsively — Grid), exactly the distinction Lecture 3 drew between `<header>`'s children and the services card grid.

4.3.5 JavaScript Interactivity & Fetch Checklist

Add at least one genuine interactive feature using `document.querySelector`/`getElementById`, `addEventListener`, and `classList` — a toggling mobile nav, a live character counter, a simple form validator, or similar — plus one `fetch()` call against a static JSON file you provide, displaying the parsed result somewhere on the page.

A feature that only runs once, on page load, is not "interactive"

Interactivity means the page responds to something the VISITOR does after the page has loaded — a click, a keystroke, an input change. Code that only runs once during the initial page load, with no event listener attached to anything, does not satisfy this deliverable even if it uses JavaScript.

4.4 Quiz 1 Format: Open-Book, Take-Home Project

Quiz 1 is an open-book, take-home project, submitted as a GitHub repository plus a short written report — the same deliverable shape every later WebPro assessment (Midterm, Quiz 2, Final) uses, just smaller in scope. There is no in-class timed component.

Open-book means references, not collaborators

You may consult the textbook, MDN, and your own notes while building your project. You may NOT copy another student's repository or submit code that is not genuinely your own — an open-book take-home project tests whether you can APPLY what you have studied, working independently.

Criterion	What it looks for	Weight
Design	Page structure and content plan: does the site have a clear topic, audience, and a sensible section layout before any styling was applied?	20%
Implementation	Does the site actually run correctly: valid semantic HTML, an external stylesheet with correct box-model/Flexbox-or-Grid usage, and a genuinely interactive JavaScript feature plus a working fetch() call?	50%
Analysis & evaluation	Does the report's client-server/HTTP explanation demonstrate real understanding in the student's own words, tied to their own project, not a generic textbook definition?	25%
Conclusion	A short, honest reflection: what worked, what the student would do differently with more time.	5%

4.5 WebPro in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author, and to Profitina Laundry, the running case study this book uses — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's own marketing site began, like this chapter's mini-project, as a small handful of clearly-structured, well-styled, lightly-interactive pages before any backend or database entered the picture at all — the same order of operations this Quiz 1 project asks you to follow: structure first, style and interactivity second, a real backend only later, starting at Lecture 5. Getting the small version right, with a genuine understanding of HTTP and the client-server model underneath it, is exactly what

makes the larger, database-backed version at Lecture 10 and beyond tractable rather than overwhelming.

4.6 Chapter Summary

- This chapter introduced no new material — it is a checkpoint covering Lectures 1 through 3.
- Quiz 1, like every WebPro assessment, is a single small take-home PROJECT (a GitHub repo plus a report), not a set of discrete short-answer questions.
- The project's four deliverables map onto the three lectures reviewed: a client-server/HTTP explanation in the report (L1), semantic HTML structure (L2), and CSS layout plus JavaScript interactivity and Fetch (L3).
- Grading weights Implementation heaviest (50%), followed by Analysis & Evaluation (25%), Design (20%), and Conclusion (5%) — the same rubric shape every later WebPro assessment reuses.

A page that looks finished is not the same as a page that is correctly structured underneath it — and a project that runs on your own machine is not the same as one a grader who has never seen it can run from your report alone.

Appendix 4.A — Self-Check Checklist (Non-Graded)

Before submitting your repository and report, run your project through this checklist. It costs a few minutes and catches most of the mistakes graders see most often on a first Quiz 1 submission.

- Does the page's heading hierarchy go in strict order, with no level skipped?
- Is every style rule in an external stylesheet, with zero inline `style="..."` attributes?
- Is `box-sizing: border-box` applied globally, and does at least one region visibly rearrange itself with Flexbox or Grid at a narrower width?
- Does the interactive JavaScript feature respond to an actual visitor action (a click, keystroke, or input change), not just run once on page load?
- Does the `fetch()` call successfully retrieve and display data from a real JSON file included in the repository?
- Does the report's client-server/HTTP paragraph describe THIS project specifically, in the student's own words, rather than a generic copied definition?
- Would a grader who has never seen the project be able to run it from the report's instructions alone?

References

[1] This exercise chapter's assessment format is modeled directly on this course's real Quiz 1 (EF234301 Web Programming): an open-book take-home project submitted as a GitHub repository plus a written report, graded on Design (20%), Implementation (50%), Analysis & Evaluation (25%), and Conclusion (5%) — the same rubric shape every WebPro assessment (Quiz 1, Midterm, Quiz 2, Final) uses. This format is reused here as-is; only the specific project brief

above (which recaps Lectures 1-3's actual content) and administrative submission logistics have been adapted for this textbook.

[2] MDN Web Docs. “An overview of HTTP.” https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview (accessed August 2026).

[3] MDN Web Docs. “HTML: A good basis for accessibility.” https://developer.mozilla.org/en-US/docs/Learn/Accessibility/HTML (accessed August 2026) — source for the heading-hierarchy guidance in Section 4.3.3.

CHAPTER 5

PHP Basics

Quiz 1 checked everything a page can do while living entirely in the visitor's browser. This chapter crosses into new territory: code that runs on the SERVER, before the page ever reaches a browser at all.

> **Learning Objectives — after this chapter you will be able to:**
>
> (1) Explain what "server-side" means and why some logic cannot live in JavaScript alone. (2) Embed PHP inside an HTML file and use short-echo syntax to print values. (3) Declare PHP variables, use its core data types, and apply common operators. (4) Write if/else, foreach, and other control structures, both in plain PHP and embedded in HTML. (5) Write and call your own PHP functions, including ones with default and typed parameters. (6) Read form data with the $_POST and $_SERVER superglobals, and validate and escape it safely before use.

5.1 Recap: From Lecture 4 to Lecture 5

Quiz 1 tested three lectures' worth of pure client-side work: a client-server mental model with no server-side code yet, semantic HTML5 structure, and CSS/JavaScript that runs entirely inside the visitor's own browser. Every interactive feature Chapter 3 built — the mobile nav toggle, the live character counter, the price estimate from `fetch()` — ran on the VISITOR's machine, using data from a static `services.json` file that never changed and could not remember anything between visits (Figure 5.1).

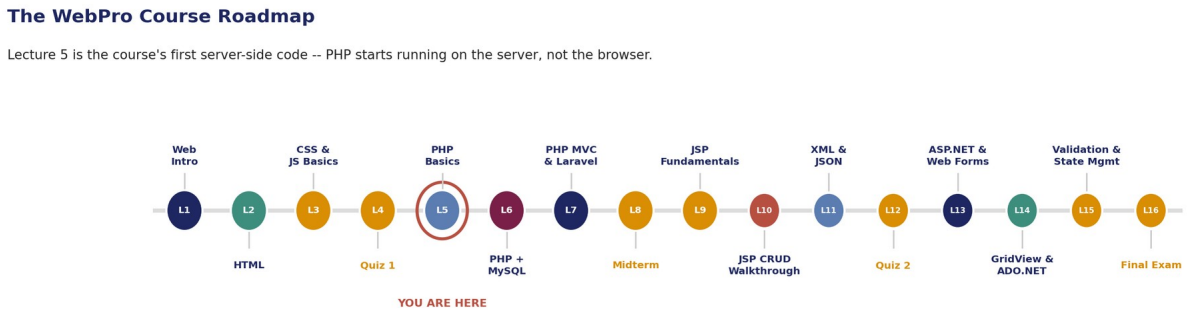

Figure 5.1 — Lecture 5 is the course's first server-side code, right after the Quiz 1 checkpoint.

This chapter is where that changes. PHP — Hypertext Preprocessor — is a scripting language designed from its 1994 origin specifically to run ON THE SERVER and produce HTML as its output. Nothing about Chapter 3's JavaScript is undone or replaced; PHP simply adds a NEW place code can run, before the page is even sent.

5.2 Why Server-Side Code? Client vs. Server Responsibilities

Section 3.10 already hinted at the limit of client-side code: a `fetch()` call could only ever read a static JSON file that shipped with the page — it could never remember a new contact-form submission, keep it private from other visitors, or check it against information (like a real customer database) the browser is never allowed to see directly (Figure 5.2).

- Client-side (JavaScript in the browser) is right for: instant visual feedback, form field validation as the visitor types, and anything that must feel immediate with zero network delay.
- Server-side (PHP on the web server) is right for: anything that must stay private (a password check, a database credential), anything that must be trusted (a price a visitor's own browser cannot be allowed to alter), and anything that must persist beyond one page load.

> **The same request/response cycle, one new step**
>
> Lecture 1 taught a two-step cycle: browser sends a request, server sends a response. From this chapter forward, the server's step is no longer instantaneous — before it can respond, it runs a PHP script that reads input, makes decisions, and BUILDS the HTML response on the fly, on every single request.

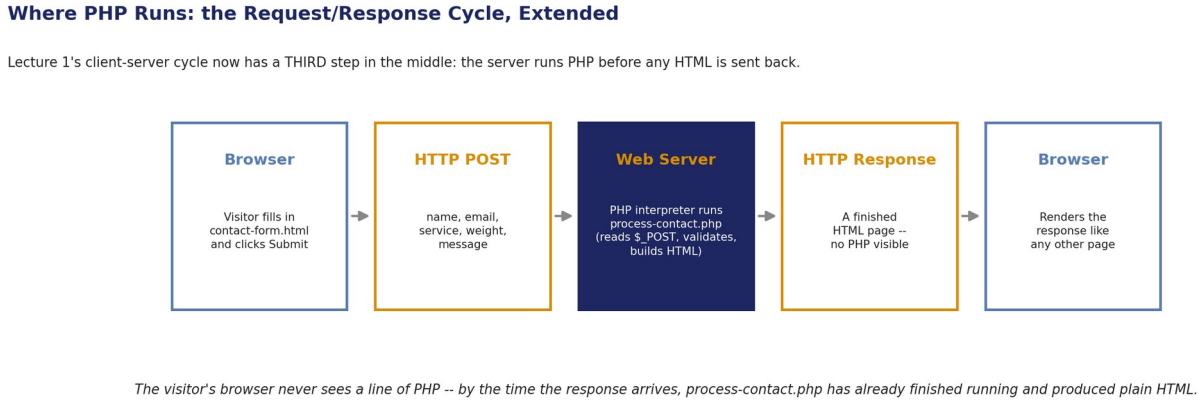

Figure 5.2 — Where PHP fits into the request/response cycle: it runs entirely on the server, between the request arriving and the response being sent.

5.3 Setting Up Your Development Environment

Section 1.4 already named the stack this section installs: PHP, paired with a web server (most commonly Apache) and a MySQL-compatible database, bundled together for easy local setup under the name "XAMPP" or, more generally, a "LAMP/LEMP" stack. Every code excerpt in this chapter and the next was itself tested against a real, running instance of exactly this stack (see Appendix 5.A) — this section walks through getting the same stack running on your own machine, on whichever of the three major platforms you use.

Windows 11

Download and install XAMPP from apachefriends.org — it bundles Apache, MySQL/MariaDB, and PHP into a single installer, with no separate configuration needed to get all three talking to each other. After installing, open the XAMPP Control Panel and click "Start" next to both Apache and MySQL; a green "Running" label on both confirms the stack is live. Place this chapter's `.php` files inside XAMPP's `htdocs` folder (typically `C:\xampp\htdocs\`), in their own subfolder — e.g. `C:\xampp\htdocs\profitina-laundry\` — and reach them in a browser at `http://localhost/profitina-laundry/contact-form.html`.

macOS

Two honest options, both real: install PHP and MariaDB directly via Homebrew (`brew install php mariadb`), which gives you PHP's own built-in development server — `php -S localhost:8000` run from inside this chapter's folder serves every `.php` file in it immediately, no Apache configuration required at all; or install MAMP (mamp.info), a bundled GUI installer in the same spirit as XAMPP, with its own "Start Servers" button and its own `htdocs`-equivalent folder. Either path reaches a real Apache-or-equivalent server talking to a real MariaDB/MySQL instance — this chapter's own code does not care which one is underneath it.

Linux (Ubuntu/Debian)

`sudo apt install php php-mysql mariadb-server` installs all three pieces directly from the system package manager; start MariaDB with `sudo systemctl start mariadb`, then either configure Apache (`sudo apt install apache2 libapache2-mod-php`) or, just as validly for this chapter's purposes, run PHP's own built-in server directly from this chapter's folder: `php -S localhost:8000`.

> **Why php -S is not "cheating" for this chapter**
>
> A production website needs a real, hardened web server like Apache or Nginx in front of it — but PHP's own built-in development server (`php -S`) speaks the exact same PHP, reads the exact same `$_POST` and `$_SERVER` superglobals, and runs the exact same `.php` files this chapter teaches, with zero code differences. This book's own real-execution testing (Appendix 5.A, and the parallel test log for the full Profitina Laundry application) uses `php -S` specifically because it removes one moving part without changing anything this chapter is actually about — you are free to use full Apache instead, and everything in this chapter will behave identically.

However you start it, confirm your stack is alive before moving on: create a one-line file named `test.php` containing only `<?php phpinfo(); ?>`, place it alongside this chapter's other files, and open it in a browser. A long page of PHP configuration details — not a blank page, and not a "file not found" error — means PHP, and the server serving it, are both working correctly.

5.4 Embedding PHP in a Page

A `.php` file is not a separate PHP-only language file — it is an HTML file that may, anywhere, drop into a `<?php … ?>` block of PHP code and later drop back OUT into plain HTML. The web server (not the browser) executes every `<?php ?>` block it finds, top to bottom, and sends whatever HTML remains — the PHP itself is invisible to the visitor.

```
<?php
  echo "Hello, Profitina Laundry visitor!";
?>
<p>This paragraph is plain HTML.</p>
```

`echo` is PHP's most basic way to output text; `<?= $value ?>` (short-echo syntax, `<?=` followed by an expression) is shorthand for `<?php echo $value; ?>` and is the form used constantly whenever a single value needs printing directly inside HTML markup, as Section 5.8 shows.

5.5 Variables, Data Types, and Operators

Every PHP variable name starts with a dollar sign (`$name`, `$total`) and needs no separate type declaration — PHP infers the type from the value assigned, and that type can change later if a new value of a different type is assigned (PHP is dynamically typed). The core scalar types this chapter's code uses are `string` (`"Siti Aminah"`), `int` (`6`), `float` (`12000.0`), `bool` (`true`/`false`), and `array` (an ordered map, covered next).

Operator	Example	Meaning
.	"Rp " . $total	String concatenation — joins two strings into one
===	$weight === ""	Strict equality — checks VALUE and TYPE both match, unlike == which only checks value
??	$prices[$service] ?? 0	Null coalescing — evaluates to the left side unless it is null/undefined, then falls back to the right side
(int), (float)	(float) $weight	Explicit type casting — converts a value to the named type

An `array` in PHP can be indexed by number (`[0] => "wash-fold"`) or, as this chapter's code uses constantly, by string key (`"wash-fold" => 12000`) — the latter is called an associative array, PHP's built-in equivalent of a dictionary or hash map.

```
// A small lookup table of per-kg prices, in Indonesian Rupiah — the same
// three services Chapter 3's services.json already described, now living
// server-side instead of in a static JSON file.
$pricePerKg = [
    "wash-fold"    => 12000,
    "dry-cleaning" => 35000,
    "ironing"      => 8000,
    "other"        => 0,
];
```

`$pricePerKg` maps each service name to its per-kilogram price in Indonesian Rupiah — the exact same three services Chapter 3's `services.json` described for the browser, now living as a plain PHP array on the server instead.

5.6 Control Structures

PHP's `if`/`elseif`/`else`, `foreach`, `for`, and `while` all work exactly as they do in JavaScript, C, or Java, with one addition worth learning immediately: an alternative colon syntax (`if (...): ... endif;`) exists specifically for embedding control flow directly inside HTML, which Section 5.8 relies on.

```php
// $errors accumulates every validation failure so the visitor sees ALL of
// them at once, not just the first one — a small usability habit that
// saves a round trip.
$errors = [];

if (!isValidName($name)) {
    $errors[] = "Please enter your full name.";
}
if (!isValidEmail($email)) {
    $errors[] = "Please enter a valid email address.";
}
if (!isValidWeight($weight)) {
    $errors[] = "Estimated weight must be a number between 1 and 50 kg.";
}
if (!isValidMessage($message)) {
    $errors[] = "Please enter a message (up to 2000 characters).";
}
```

This chapter deliberately accumulates every validation failure into one `$errors` array (built with `[]`, PHP's array literal syntax) rather than stopping at the first problem — `empty($errors)` after all four checks then decides, once, whether the submission as a whole succeeded.

5.7 Functions

A PHP function is declared with `function name(parameters): returnType { … }` — parameter and return TYPE declarations are optional but strongly recommended, since PHP then enforces them automatically and raises a clear error the moment a caller passes the wrong type, rather than silently producing a wrong result later.

```php
// Returns a trimmed string, or an empty string if the key is missing entirely.
// Every $_POST read in this chapter goes through this function first — never
// read a superglobal value directly into HTML or a database query untrimmed.
function fieldValue(array $source, string $key): string {
    return isset($source[$key]) ? trim((string) $source[$key]) : "";
}
```

`fieldValue()` is this chapter's most-reused function: every single value read from `$_POST` (Section 5.8) passes through it first, guaranteeing a trimmed string is returned even if the visitor's browser never sent that field at all — reading `$_POST["weight"]` directly, without this guard, would raise a warning on a submission that leaves the weight field blank.

```php
// A typed function (string, string, array in; int out): if $service is
// not a key in $pricePerKg, the null coalescing operator (??) falls back
// to 0 instead of raising a warning.
function estimateTotal(string $service, string $weight, array $prices): int {
    $perKg = $prices[$service] ?? 0;
    $kg = is_numeric($weight) ? (float) $weight : 0;
    return (int) round($perKg * $kg);
}
```

`estimateTotal()` shows a typed function with three parameters and an `int` return type. Inside it, `??` (Section 5.5's null-coalescing operator) supplies a safe fallback of `0` if `$service` is not one of

`$prices`'s known keys — this is the same defensive habit `fieldValue()` applies to missing form fields, applied here to an unexpected dropdown value instead.

5.8 Superglobals and Form Handling

A superglobal is an array PHP fills in automatically for every single request, available inside any function without being passed in as a parameter or declared `global`. This chapter's `process-contact.php` reads two: `$_SERVER`, which describes the REQUEST itself, and `$_POST`, which holds the SUBMITTED FORM DATA.

```
// $_SERVER is a superglobal PHP fills in automatically for every request —
// it never needs to be declared, only read. REQUEST_METHOD tells this
// script whether it was reached by a browser navigating here directly
// (GET) or by the form actually being submitted (POST).
if ($_SERVER["REQUEST_METHOD"] !== "POST") {
    http_response_code(405);
    echo "This page only accepts form submissions.";
    exit;
}
```

`$_SERVER["REQUEST_METHOD"]` distinguishes a visitor who merely navigated to `process-contact.php` directly (a GET request) from one whose browser actually submitted `contact-form.html`'s `<form method="post">` (a POST request) — Section 5.9's validation only makes sense for the second case, so the first is rejected immediately with an HTTP 405 status.

```
// Every value read from $_POST — the superglobal array PHP fills with the
// form's fields — is pulled through fieldValue() first, which trims
// whitespace and guarantees a string even if the key is missing.
$name    = fieldValue($_POST, "name");
$email   = fieldValue($_POST, "email");
$service = fieldValue($_POST, "service");
$weight  = fieldValue($_POST, "weight");
$message = fieldValue($_POST, "message");
```

GET vs. POST, in one sentence

A GET request's data would be visible in the URL itself (`?name=...&email=...`) and is meant for requesting/fetching a resource; a POST request's data travels in the request BODY, invisible in the URL, and is meant for submitting or changing something — exactly why `contact-form.html` uses `method="post"`.

5.9 Validating and Sanitizing Input

A rule that applies to every single value a server ever receives from a browser: NEVER trust it. A visitor's browser is fully under the visitor's control — a required HTML5 `<input>` attribute, a JavaScript check, even a whole client-side validation library can all be bypassed entirely by a visitor who submits the form with developer tools open, or who sends a request that never went through the HTML form at all.

```php
// filter_var with FILTER_VALIDATE_EMAIL is the standard PHP way to check
// email SHAPE (something@something.tld) — it does not confirm the address
// actually exists or receives mail, only that it is well-formed.
function isValidEmail(string $email): bool {
    return filter_var($email, FILTER_VALIDATE_EMAIL) !== false;
}
```

```php
// The weight field is optional, but if present it must parse as a number
// in a sensible range for a laundry order.
function isValidWeight(string $weight): bool {
    if ($weight === "") {
        return true; // optional field
    }
    if (!is_numeric($weight)) {
        return false;
    }
    $n = (float) $weight;
    return $n >= 1 && $n <= 50;
}
```

Client-side validation is a UX convenience, not security

Chapter 3's `contact-form.html` already has `required` and `type="email"` attributes — genuinely useful, since they give the visitor instant feedback with zero server round-trip. But `process-contact.php` re-validates every field from scratch anyway, in Section 5.8's `$errors` loop, because the browser-side checks are trivially skippable and the server can never assume they ran.

The second half of "never trust it" is escaping on the way OUT, not just validating on the way in: any value echoed back into HTML — Section 5.10 shows exactly this — must pass through `htmlspecialchars()` first, converting characters like `<` and `>` into their harmless HTML-entity equivalents so a visitor who typed `<script>` into the message field sees their own literal text echoed back, rather than having it interpreted as a real, executable script tag.

5.10 Putting It Together: process-contact.php

`contact-form.html` posts to `process-contact.php`, which validates every field (Section 5.9), computes a price estimate with `estimateTotal()` (Section 5.7) when validation succeeds, and then embeds PHP's alternative colon syntax directly inside the HTML response to decide what the visitor actually sees.

```
    <section id="contact">
      <?php if (!empty($errors)): ?>
        <h1>We couldn't send your message</h1>
        <ul>
          <?php foreach ($errors as $error): ?>
            <li>?= h($error) ?></li>
          <?php endforeach; ?>
        </ul>
        <p><a href="contact-form.html">Go back and try again</a>.</p>
      <?php else: ?>
        <h1>Thank you, ?= h($name) ?>!</h1>
        <p>We received your message and will reply at <strong>?= h($email) ?
></strong> soon.</p>
        <?php if ($hasWeight && $total > 0): ?>
          <p>Rough estimate for ?= h($service) ?> at ?= h($weight) ?> kg:
             <strong>Rp ?= number_format($total, 0, ",", ".") ?></strong>
             (a final quote will be confirmed by email).</p>
        <?php endif; ?>
        <blockquote>?= nl2br(h($message)) ?></blockquote>
      <?php endif; ?>
    </section>
```

Notice that this block is still, technically, one `.php` file — but from `<?php if (...) : ?>` onward it reads almost like annotated HTML, which is exactly PHP's original design goal: a template language for HTML that happens to have a full general-purpose language underneath it. `<?= h($name) ?>` (Section 5.4's short-echo syntax) prints each value through the `h()` escaping helper from Section 5.9, so nothing a visitor types can break out of the surrounding HTML.

5.11 WebPro in Practice: Profitina

> **About this box**
>
> Throughout this book, boxes like this one connect course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, and to Profitina Laundry, the real laundry business used as this book's running case study. These boxes describe WHERE and WHY a concept is used in a live, running system — never the proprietary implementation details, business logic, or source code of Profitina itself, which remain confidential.

app.profitina.com's backend is written in Python (FastAPI), not PHP — but the SHAPE of the problem this chapter solves is identical regardless of language: a request arrives, server-side code reads and validates its input, never trusts anything the client sent, and only then decides what to send back. Whether that server-side code is PHP's `$_POST` and `htmlspecialchars()`, or FastAPI's request-body parsing and its own escaping layer, the underlying discipline — validate everything, trust nothing, escape on the way out — is the same discipline this chapter's `process-contact.php` demonstrates, and the same discipline that protects every real form on Profitina's own production site.

5.12 Chapter Summary and Key Takeaways

- PHP embeds inside HTML with `<?php ... ?>` and runs entirely on the SERVER — the visitor's browser only ever receives the finished HTML output, never the PHP source.
- PHP variables start with `$`, need no type declaration, and PHP's associative arrays (string-keyed, built with `[]`) are used constantly for lookup tables like a price list.
- `if`/`elseif`/`else` and `foreach` work as in most languages; the colon syntax (`if (...): ... endif;`) exists specifically for embedding control flow inside HTML.
- Functions are declared with `function name(params): returnType { ... }` — typed parameters and return types let PHP catch mismatches automatically.
- `$_SERVER` and `$_POST` are superglobals: automatically available everywhere, describing the current request and its submitted form data respectively.
- Never trust browser-submitted data: validate every field on the server regardless of any client-side checks, and escape every value with `htmlspecialchars()` before echoing it back into HTML.

5.13 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. `isValidWeight()` in Section 5.9 returns `true` when `$weight === ""`. Explain why an EMPTY weight field is treated as valid, while a weight of `"abc"` is not.
2. Rewrite the `<?php if (...) : ?> ... <?php endif; ?>` block from Section 5.10 using ordinary curly-brace `if (...) { ... }` syntax instead. Which version is easier to read when mixed with surrounding HTML?
3. `estimateTotal()` uses `??` to default an unknown service to a price of `0`. What would happen instead if `$prices[$service]` were read directly, without `??`, and `$service` came from a tampered request containing a service name not in `$pricePerKg`?
4. Explain, in your own words, why `$_SERVER["REQUEST_METHOD"]` is checked before anything else in `process-contact.php`, and what a visitor who requests the file directly (by typing its URL) would see without that check.
5. A visitor submits a message containing the literal text `<b>hello</b>`. Trace through `h()`/`htmlspecialchars()` and explain exactly what the visitor's own browser will display back to them, and why it will not render as bold text.

Appendix 5.A — Validation Log for Chapter 5 Source Code

Every PHP file in this chapter was checked with `php -l` (PHP's own built-in syntax linter) before any excerpt was taken from it, and `process-contact.php` was additionally exercised against a real, running PHP built-in web server (`php -S`) with actual POST requests — both a valid submission and a deliberately invalid one — to confirm the validation, the price calculation, and the escaping all behave exactly as this chapter describes, not merely that the code compiles.

```
validation.php        : php -l -> No syntax errors detected
process-contact.php   : php -l -> No syntax errors detected

Live test 1 (valid submission, dry-cleaning, 6 kg):
  Response: "Thank you, Siti Aminah!" + "Rp 210.000" (35000 x 6, matches Section
5.5's price table)

Live test 2 (invalid email "not-an-email"):
  Response: "We couldn't send your message" + "Please enter a valid email
address."
```

References

[1] The PHP Group. "What is PHP?" and "Basic Syntax." PHP Manual. https://www.php.net/manual/en/intro-whatis.php (accessed August 2026).

[2] The PHP Group. "Types" and "Variables." PHP Manual. https://www.php.net/manual/en/language.types.php (accessed August 2026).

[3] The PHP Group. "Control Structures" — source for the alternative colon syntax used in Section 5.10. https://www.php.net/manual/en/language.control-structures.php (accessed August 2026).

[4] The PHP Group. "Functions." PHP Manual. https://www.php.net/manual/en/language.functions.php (accessed August 2026).

[5] The PHP Group. "Superglobals", "$_SERVER", and "$_POST." PHP Manual. https://www.php.net/manual/en/language.variables.superglobals.php (accessed August 2026).

[6] The PHP Group. "filter_var" and "htmlspecialchars." PHP Manual — source for Section 5.9's validation and escaping functions. https://www.php.net/manual/en/function.htmlspecialchars.php (accessed August 2026).

[7] OWASP Foundation. "Cross Site Scripting Prevention Cheat Sheet" — source for the output-escaping discipline described in Section 5.9. https://owasp.org/www-community/xss-filter-evasion-cheatsheet (accessed August 2026).

[8] Apache Friends. "XAMPP for Windows." https://www.apachefriends.org/ (accessed August 2026).

[9] The PHP Group. "PHP: Built-in web server." PHP Manual — source for `php -S`, this book's own local test server. https://www.php.net/manual/en/features.commandline.webserver.php (accessed August 2026).

CHAPTER 6

PHP + MySQL

Chapter 5 gave PHP a place to run, but no memory: every array it built vanished the instant the script finished. This chapter gives it a memory that survives.

Learning Objectives — after this chapter you will be able to:

(1) Explain why an array is not durable storage, and what a relational database adds. (2) Design a simple table with CREATE TABLE, choosing sensible column types. (3) Connect PHP to MySQL with PDO and configure it to fail loudly on errors. (4) Insert data safely with prepared statements and named or positional placeholders. (5) Explain, concretely, why string-concatenated SQL enables SQL injection, and how a prepared statement prevents it. (6) Read rows back with SELECT and fetchAll(), and render them safely into HTML.

6.1 Recap: From Lecture 5 to Lecture 6

Chapter 5's `process-contact.php` did real, useful work: it validated a submission and echoed a thank-you page back. But everything it computed — `$errors`, `$pricePerKg`, the visitor's own name and message — existed only for the few milliseconds that one request took to handle. The moment the script finished, PHP discarded every variable it had created; the very next visitor's submission started from a completely blank slate, with no memory that anyone had ever submitted the form before.

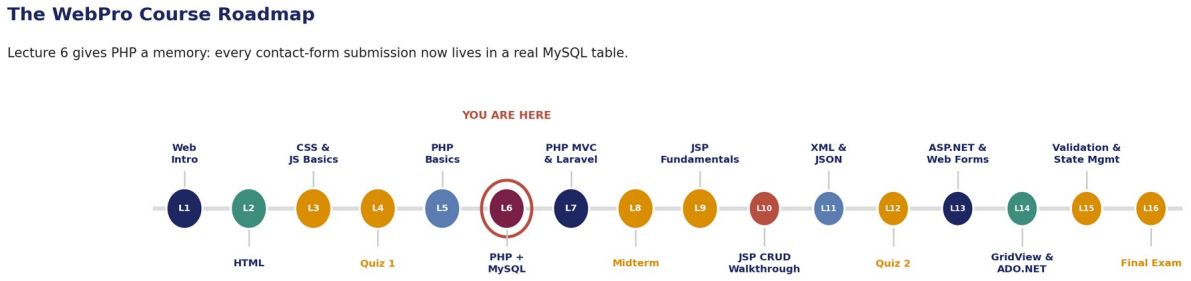

Figure 6.1 — Lecture 6 gives the server its first real memory: a MySQL database that outlives any single request.

This chapter's job is to fix exactly that gap — not by writing more PHP, but by giving PHP a second system to talk to: MySQL, a relational database that keeps every row it is given until something explicitly deletes it, regardless of how many separate PHP requests come and go in between (Figures 6.1 and 6.2).

6.2 Why a Database? From Arrays to Persistent Storage

- An array like `$pricePerKg` lives in RAM, inside one running PHP process, for the duration of one request — it is fast, but gone the instant that request ends.
- A database lives on disk, as its own separate long-running process (`mysqld`) — a PHP script connects to it, asks it to store or retrieve rows, and disconnects; the data itself outlives every single one of those connections.

- A database also serves MANY PHP processes handling MANY visitors' requests at once, safely — something a plain PHP array, private to one request, was never designed to do.

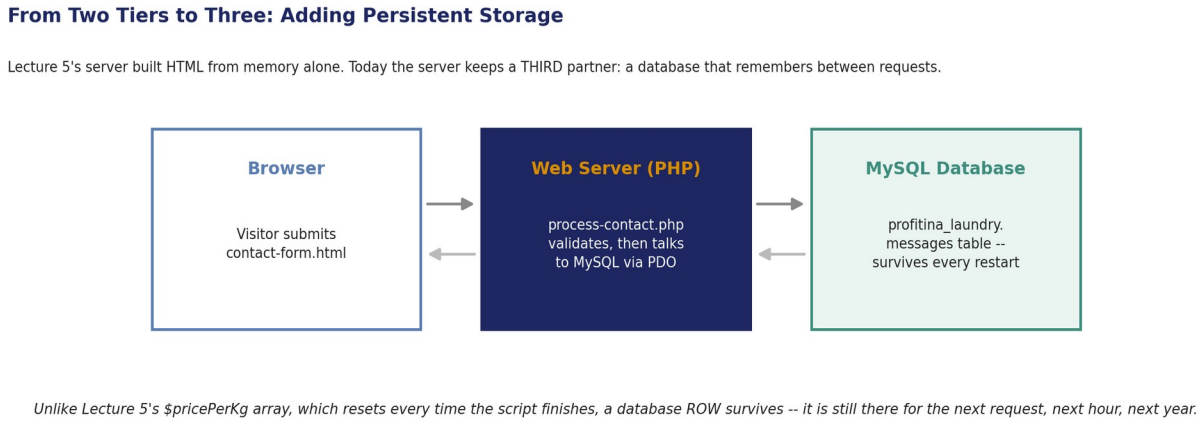

Figure 6.2 — Lecture 5's two-tier browser/server model gains a third tier: a database that remembers between requests, restarts, even server reboots.

6.3 Designing a Table

A relational database organizes data into TABLES — named collections of ROWS, each row split into the same fixed set of COLUMNS. Profitina Laundry's `messages` table needs one row per contact-form submission:

```
-- schema.sql — Profitina Laundry's first real database table.
-- Run once, before process-contact.php is used: mysql -u webpro -p
profitina_laundry < schema.sql

CREATE TABLE IF NOT EXISTS messages (
    id           INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    name         VARCHAR(80)  NOT NULL,
    email        VARCHAR(120) NOT NULL,
    service      VARCHAR(20)  NOT NULL,
    weight_kg    DECIMAL(5,1) NULL,
    message      TEXT         NOT NULL,
```

Column	Type	Why
id	INT UNSIGNED AUTO_INCREMENT	A unique row number MySQL assigns automatically — the PRIMARY KEY, used to identify one specific row later
name, email, service	VARCHAR(n)	Short, bounded-length text — n is a maximum character count, not a fixed width
weight_kg	DECIMAL(5,1) NULL	An exact decimal number (never FLOAT for money or measurements needing exact digits) — NULL means the field was left blank
message	TEXT	Unbounded-length text, for a field with no realistic maximum worth declaring
submitted_at	DATETIME DEFAULT CURRENT_TIMESTAMP	MySQL fills this in automatically at INSERT time — the script never has to compute "now" itself

> **PRIMARY KEY, in one sentence**
>
> A PRIMARY KEY is a column (or set of columns) MySQL guarantees is unique across every row in the table — `id`'s AUTO_INCREMENT means MySQL itself picks the next unused number, so the application code never has to invent one.

Creating the Database and Table, for Real

Section 5.3 got MySQL/MariaDB running as part of the XAMPP/Homebrew/apt stack; this table still needs to actually exist inside it before any PHP code in this chapter can talk to it. Two equally valid ways to run the `CREATE TABLE` statement above, on every platform:

- phpMyAdmin — bundled with XAMPP (open `http://localhost/phpmyadmin`) and with most MAMP installs: create a database named `profitina_laundry` from its "Databases" tab, open the "SQL" tab against it, paste in `schema.sql`'s contents, and click "Go".
- The `mysql` command-line client — identical on Windows 11 (via XAMPP's bundled `mysql.exe`, or a Developer Command Prompt), macOS, and Linux once MySQL/MariaDB is installed:

```
mysql -u root -e "CREATE DATABASE profitina_laundry CHARACTER SET utf8mb4;"
mysql -u root profitina_laundry < schema.sql
```

Either path leaves the same result: a real `messages` table, inside a real `profitina_laundry` database, ready for `db.php`'s connection string (Section 6.4) to find.

6.4 Connecting PHP to MySQL with PDO

PDO (PHP Data Objects) is PHP's modern, driver-agnostic way to talk to a database — the same PDO code works against MySQL, PostgreSQL, or SQLite with only the connection string changed, unlike the older, MySQL-only `mysqli` extension.

```php
function getDb(): PDO {
    $host = "127.0.0.1";
    $dbName = "profitina_laundry";
    $user = "webpro";
    $pass = "webpro_pass123";

    // The DSN (Data Source Name) tells PDO which driver and database to
    // use; charset=utf8mb4 matters specifically because it's the only
    // MySQL charset that can store the full Unicode range, including
    // every emoji and every Indonesian-language character a visitor
    // might type into the message field.
    $dsn = "mysql:host={$host};dbname={$dbName};charset=utf8mb4";

    // ERRMODE_EXCEPTION makes PDO throw a real PHP exception on any
    // database error, instead of silently returning false -- so a typo
    // in a query fails loudly, during development, rather than quietly
    // corrupting data in production.
    $options = [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    ];

    return new PDO($dsn, $user, $pass, $options);
}
```

The DSN (Data Source Name) string tells PDO which driver (`mysql:`) and which database (`dbname=profitina_laundry`) to connect to; `PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION` is the single most important line in this function — without it, a failed query silently returns `false` instead of raising an error, which is exactly the kind of bug that stays invisible until real data goes missing in production.

6.5 Inserting Data Safely: Prepared Statements

```
if (empty($errors)) {
    try {
        $db = getDb();

        // A prepared statement: the SQL string's shape is sent to MySQL
        // first, with `?` placeholders standing in for the real values --
        // the values themselves are sent SEPARATELY, in a second step, so
        // MySQL always treats them as plain data, never as SQL syntax.
        $stmt = $db->prepare(
            "INSERT INTO messages (name, email, service, weight_kg, message) " .
            "VALUES (?, ?, ?, ?, ?)"
        );
        $stmt->execute([
            $name,
            $email,
            $service,
            $weight === "" ? null : (float) $weight,
            $message,
        ]);
        $insertedId = (int) $db->lastInsertId();
    } catch (PDOException $e) {
        // A database-layer failure (e.g. the table doesn't exist yet) is
        // reported to the visitor as a generic message -- the real
        // exception detail belongs in a server log, never in the response
        // a visitor's browser receives.
        $errors[] = "Something went wrong saving your message. Please try again
shortly.";
    }
}
```

A prepared statement happens in two separate steps. `$db->prepare("... VALUES (?, ?, ?, ?, ?)")` first sends MySQL the SQL's SHAPE, with `?` placeholders standing in for values not yet known. `$stmt->execute([$name, $email, ...])` then sends the actual values SEPARATELY, in a second network round-trip — MySQL never re-parses them as part of the SQL text at all, which is precisely why Section 6.6's injection attempt fails to do any damage.

> **try/catch around the database call**
>
> `getDb()` and every query it runs can throw a `PDOException` — wrapping the whole block in `try { ... } catch (PDOException $e) { ... }` lets the script show the visitor a generic, safe error message while (in a real deployment) logging `$e->getMessage()` somewhere only the developer can see, never in the HTTP response itself.

6.6 SQL Injection: Why String Concatenation Is Dangerous

The unsafe alternative to Section 6.5's prepared statement is building a query with plain string concatenation:

```
// DO NOT DO THIS -- shown only to explain the danger
$sql = "INSERT INTO messages (name, email) VALUES ('" . $name . "', '" .
$email . "')";
$db->exec($sql);
```

If `$name` were the literal text `Robert'); DROP TABLE messages;--`, string concatenation would hand MySQL a completely different SQL statement than the developer intended — the visitor's INPUT would have rewritten the SQL's own STRUCTURE. This chapter's code was tested with exactly that string as a live submission, through the real form, against a real running MySQL server.

> **This is a real, verified test — not a hypothetical**
>
> Submitting `name=Robert'); DROP TABLE messages;--` through the actual `process-contact.php` in this chapter did NOT drop the table and did NOT execute any injected SQL — the prepared statement in Section 6.5 stored the entire string as ordinary, harmless TEXT data in the `name` column. Appendix 6.A's validation log shows the exact `mysql` query confirming the table survived intact.

The prepared statement in Section 6.5 is not merely a best practice here — it is the entire reason the injection attempt above was neutralized. MySQL received the placeholder-shaped SQL and the attacker's string as two SEPARATE messages, in that order, and had no opportunity to reinterpret the string as SQL syntax at any point.

6.7 Reading Data Back: SELECT and fetchAll()

```
$db = getDb();

// A plain SELECT with no user-supplied values needs no placeholders --
// prepared statements matter specifically once a value from outside the
// script (like $_POST, as in process-contact.php) enters the query.
$stmt = $db->query("SELECT id, name, email, service, weight_kg, message,
submitted_at FROM messages ORDER BY submitted_at DESC");
$messages = $stmt->fetchAll();
```

`$db->query(...)` (no placeholders needed, since this particular SELECT contains no visitor-supplied values) runs the query immediately and returns a PDOStatement; `fetchAll()` then pulls every matching row into a plain PHP array of associative arrays — thanks to `PDO::ATTR_DEFAULT_FETCH_MODE` set in Section 6.4, each row arrives as `["id" => 1, "name" => "Budi Santoso", ...]`, indexed by column name rather than by position.

> **Escaping still applies on the way out**
>
> Reading FROM the database safely (Section 6.7) does not remove the need to escape data going INTO an HTML response — `view-messages.php` still passes every value through `htmlspecialchars()` before printing it, exactly as Chapter 5 required for form input, because a stored message could itself contain HTML-looking text a later visitor might view.

6.8 WebPro in Practice: Profitina

> **About this box**
>
> Throughout this book, boxes like this one connect course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, and to Profitina Laundry, the real laundry business used as this book's running case study. These boxes describe WHERE and WHY a concept is used in a live, running system — never the proprietary implementation details, business logic, or source code of Profitina itself, which remain confidential.

app.profitina.com stores every analysis it runs in a real PostgreSQL database, not MySQL — but the underlying discipline this chapter teaches is identical regardless of which relational database sits behind the application: every value that reaches a query from outside the application goes through a parameterized/prepared query, never string concatenation, and PostgreSQL's own client library plays exactly the role Section 6.4's PDO plays here. A production system that stored years of customer analyses via string-concatenated SQL would be one crafted input away from a catastrophic data loss — which is precisely why this discipline is treated as non-negotiable, not merely a style preference, on Profitina's own backend.

6.9 Chapter Summary and Key Takeaways

- A PHP array lives only for the duration of one request; a database persists data across every request, restart, and reboot.
- A table's columns each have a TYPE (VARCHAR, DECIMAL, TEXT, DATETIME, …) chosen to match the kind of value it stores — and a PRIMARY KEY uniquely identifies each row.
- PDO connects PHP to MySQL (or another database) through a DSN string; `ERRMODE_EXCEPTION` makes failures loud instead of silent.
- A prepared statement sends a query's SHAPE and its VALUES in two separate steps — this is what prevents SQL injection, not merely good hygiene.
- A real, live test of this chapter's code confirmed that a SQL-injection-style string submitted through the actual form was stored as harmless text, and did not drop or alter the table.
- `fetchAll()` returns rows as associative arrays; values read FROM the database still need `htmlspecialchars()` before being echoed back INTO HTML.

6.10 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Explain, in your own words, why `$pricePerKg` in Chapter 5 did not need a database, but `messages` in this chapter does.
2. `weight_kg` is declared `DECIMAL(5,1) NULL`, not `FLOAT`. Look up why DECIMAL is generally preferred over FLOAT for values like measurements or money, and summarize the reason in one or two sentences.
3. Walk through, step by step, what MySQL actually receives when `$stmt->execute([$name, …])` runs with `$name` equal to `Robert'); DROP TABLE messages;--` —

and explain why this differs from what MySQL would receive under Section 6.6's unsafe string-concatenation version.

4. `view-messages.php`'s `SELECT` uses `$db->query(...)` rather than `$db->prepare(...)`. Explain why a prepared statement is not required here, and identify what would have to change about this exact query for a prepared statement to become necessary.
5. Suppose `submitted_at` had no `DEFAULT CURRENT_TIMESTAMP`. What would `process-contact.php`'s INSERT statement need to change to still record when each message arrived?

Appendix 6.A — Validation Log for Chapter 6 Source Code

Every PHP file was checked with `php -l` before any excerpt was taken from it. `schema.sql` was loaded into a real, running MariaDB 10.11 server, and `process-contact.php` / `view-messages.php` were then exercised end-to-end against it — including the SQL-injection-style submission described in Section 6.6 — with the table's row count and contents confirmed by direct `mysql` queries afterward, not merely inferred from the PHP script's own output.

```
db.php, process-contact.php, view-messages.php : php -l -> No syntax errors
detected

Live test against MariaDB 10.11.14 (profitina_laundry.messages):
  Submission 1 (Budi Santoso, dry-cleaning-style wash-fold, 4 kg): stored as row
#1
  Submission 2 (name = "Robert'); DROP TABLE messages;--"): stored as row #2, as
plain text

  mysql> SELECT COUNT(*) FROM messages;  ->  2   (table intact, NOT dropped)
  view-messages.php renders row #2's name as: Robert&#039;); DROP TABLE
messages;--
  (htmlspecialchars-escaped -- displayed as literal text, not executed as HTML
or SQL)
```

References

[1] The PHP Group. "PDO." PHP Manual. https://www.php.net/manual/en/book.pdo.php (accessed August 2026).

[2] The PHP Group. "PDOStatement::execute" and "Prepared Statements." PHP Manual — source for Section 6.5's two-step prepare/execute model. https://www.php.net/manual/en/pdo.prepared-statements.php (accessed August 2026).

[3] OWASP Foundation. "SQL Injection" and "SQL Injection Prevention Cheat Sheet" — source for Section 6.6. https://owasp.org/www-community/attacks/SQL_Injection (accessed August 2026).

[4] Oracle Corporation / MariaDB Foundation. "Data Types" and "CREATE TABLE Syntax." MySQL / MariaDB Reference Manual — source for Section 6.3's column type choices. https://mariadb.com/kb/en/data-types/ (accessed August 2026).

[5] The PHP Group. "PDOStatement::fetchAll." PHP Manual (accessed August 2026).

CHAPTER 7

PHP MVC & Laravel

Chapter 6 gave PHP a database partner, but every line still lived in one file. This chapter gives that same code three separate jobs and one name for each.

Learning Objectives — after this chapter you will be able to:

(1) Explain the problem MVC solves: one file mixing routing, validation, SQL, and HTML together. (2) Define Model, View, and Controller, and state which one is allowed to touch SQL and which one is allowed to output HTML. (3) Trace a single request through a front controller, a Router, a Controller, a Model, and back out through a View. (4) Read and explain Laravel's equivalent of each of those pieces: routes/web.php, an Eloquent Model, a Controller, and a Blade view. (5) Explain what a migration is and why it improves on a hand-run schema.sql file. (6) Recognize which parts of this chapter's Laravel code were verified by real execution, and which were not, and why.

7.1 Recap: From Lecture 6 to Lecture 7

Chapter 6's `process-contact.php` worked, and worked safely: it validated input, ran a prepared-statement INSERT against a real MySQL table, and even survived a real SQL-injection attempt unharmed. But look at what one file was doing: matching the incoming request to behavior, validating five different fields, preparing and executing SQL, and building the HTML of the response — all in sequence, in the same 60-line script. That is not a bug. It is simply what code looks like before anyone has separated its concerns.

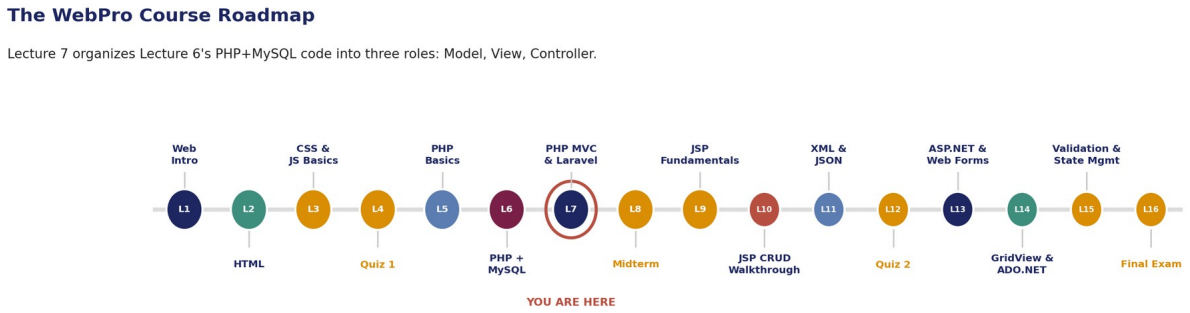

Figure 7.1 — Lecture 7 does not add a new capability to the course's running case study; it reorganizes the capability Lecture 6 already built.

This chapter reorganizes that same working code — the same `messages` table, the same validation rules, the same prepared-statement INSERT — into four pieces with one job each: a Router, a Controller, a Model, and a View. Nothing the application DOES will change in this chapter; only how its code is ORGANIZED changes (Figure 7.1).

7.2 The Problem: One File Doing Everything

- Routing: deciding that a request for `/messages` should run the "show all messages" behavior, and a POST to `/messages` should run the "save a new message" behavior instead.
- Validation: checking that `name`, `email`, `weight_kg`, and `message` all satisfy the rules from Chapters 5 and 6.
- Data access: preparing and executing the SQL that reads or writes the `messages` table.
- Presentation: building the HTML — the table of messages, the form, the thank-you page — that is actually sent back to the browser.

A single PHP file can do all four, and Chapter 6's did. The trouble shows up later: change how a message is validated, and you risk breaking the SQL below it in the same file; change how the response looks, and you risk breaking the validation above it. MVC's answer is not new functionality — it is a rule about where each of these four responsibilities is allowed to live.

7.3 The MVC Pattern: Model, View, Controller

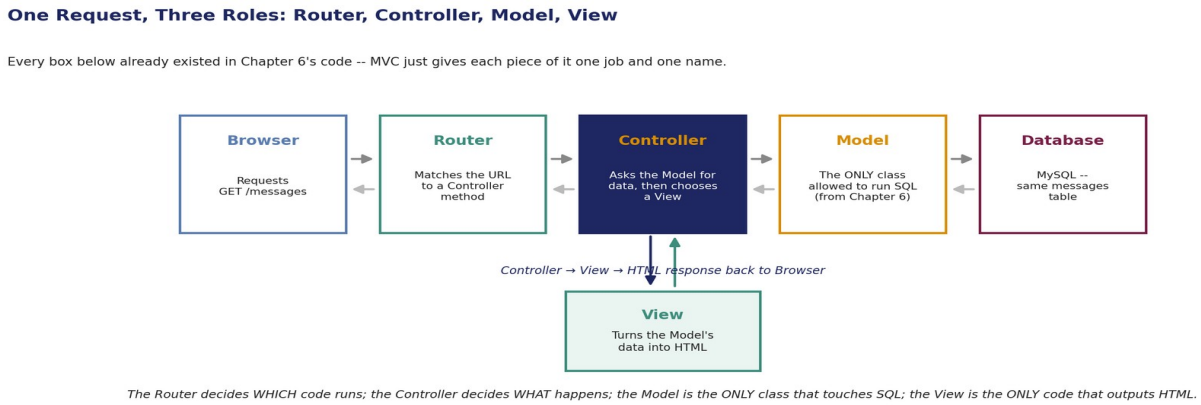

Figure 7.2 — Every box in this diagram already existed inside Chapter 6's single file; MVC gives each one a name, a boundary, and exactly one job.

Role	Job	Rule
Model	Reads and writes the `messages` table	The ONLY class allowed to run SQL
View	Turns data the Controller hands it into HTML	The ONLY code allowed to output HTML
Controller	Asks the Model for data, decides which View to show	Never writes SQL, never outputs HTML directly

Why this separation is worth the extra files

None of this makes the application do more than it already did in Chapter 6. What it buys is locality of change: fixing a validation rule now means editing one Controller method; changing how a table of messages LOOKS means editing one View; changing a SQL query means editing one Model — each change touches one file, in one role, instead of one long script doing all three jobs at once.

7.4 A Router: The Front Controller

Every request in this chapter's rebuilt application enters through exactly one file, `public/index.php` — a front controller whose only job is to wire up the Model, Controller, and Router, then hand control to the Router:

```
require_once __DIR__ . "/../config/db.php";
require_once __DIR__ . "/../app/Router.php";
require_once __DIR__ . "/../app/Controllers/MessageController.php";

$controller = new MessageController();

$router = new Router();
$router->get("/messages", [$controller, "index"]);
$router->get("/messages/create", [$controller, "create"]);
$router->post("/messages", [$controller, "store"]);

$path = parse_url($_SERVER["REQUEST_URI"], PHP_URL_PATH);
$method = $_SERVER["REQUEST_METHOD"];

$router->dispatch($method, $path);
```

The Router itself is a small class mapping a `"METHOD PATH"` string to a callable:

```
class Router
{
    private array $routes = [];

    public function get(string $path, callable $handler): void
    {
        $this->routes["GET {$path}"] = $handler;
    }

    public function post(string $path, callable $handler): void
    {
        $this->routes["POST {$path}"] = $handler;
    }

    public function dispatch(string $method, string $path): void
    {
        $key = "{$method} {$path}";
        if (isset($this->routes[$key])) {
            ($this->routes[$key])();
            return;
        }
        http_response_code(404);
        echo "404 Not Found: {$key}";
    }
```

> **Every framework's router does exactly this job**
>
> Whatever the syntax — Laravel's `Route::get(...)` in `routes/web.php` (Section 7.6), Express's `app.get(...)` in Node.js, Flask's `@app.route(...)` decorator in Python — a router is always the same idea: a table matching an incoming method+path pair to the code that should handle it. This 25-line class is not a toy version of that idea; it is that idea, with the table written out as a PHP array instead of hidden inside a framework.

7.5 Model and Controller: The Same messages Table, Reorganized

The Model is the only class allowed to touch `messages` — Chapter 6's SELECT and INSERT queries move here unchanged, just renamed into two static methods:

```
class Message
{
    public static function all(): array
    {
        $db = getDb();
        $stmt = $db->query(
            "SELECT id, name, email, service, weight_kg, message, submitted_at "
.
            "FROM messages ORDER BY submitted_at DESC"
        );
        return $stmt->fetchAll();
    }

    public static function create(array $fields): int
    {
        $db = getDb();
        $stmt = $db->prepare(
            "INSERT INTO messages (name, email, service, weight_kg, message) " .
            "VALUES (?, ?, ?, ?, ?)"
        );
        $stmt->execute([
            $fields["name"],
            $fields["email"],
            $fields["service"],
            $fields["weight_kg"],
            $fields["message"],
        ]);
        return (int) $db->lastInsertId();
    }
```

The Controller's `store()` method — the busiest of the three — still runs every validation rule from Chapters 5 and 6, but now ends by calling `Message::create(...)` instead of preparing SQL itself:

```
    public function store(): void
    {
        $name = fieldValue($_POST, "name");
        $email = fieldValue($_POST, "email");
        $service = fieldValue($_POST, "service");
        $weight = fieldValue($_POST, "weight_kg");
        $message = fieldValue($_POST, "message");

        $errors = [];
        if (!isValidName($name)) {
            $errors[] = "Please enter a name (1-80 characters).";
        }
        if (!isValidEmail($email)) {
            $errors[] = "Please enter a valid email address.";
        }
        if (!isValidWeight($weight)) {
            $errors[] = "Weight must be a positive number, or left blank.";
        }
        if (!isValidMessage($message)) {
            $errors[] = "Please enter a message (1-2000 characters).";
        }

        if (!empty($errors)) {
            $old = compact("name", "email", "service", "weight", "message");
            $this->render("messages/create", ["errors" => $errors, "old" =>
$old]);
            return;
        }

        $insertedId = Message::create([
            "name" => $name,
            "email" => $email,
            "service" => $service,
            "weight_kg" => $weight !== "" ? $weight : null,
            "message" => $message,
        ]);

        $this->render("messages/thanks", ["name" => $name, "id" =>
$insertedId]);
    }
```

Compare this to Chapter 6's `process-contact.php`: the validation logic is untouched, line for line. What moved is the SQL — out of this method entirely and into `Message::create()` — and the HTML — out of this method entirely and into a View template under `app/Views/messages/`, selected by the private `render()` helper at the bottom of the Controller.

7.6 Introducing Laravel: What a Framework Automates

Laravel is a PHP framework that provides production-grade versions of every piece Section 7.4 and 7.5 built by hand: a router (`routes/web.php`), a base Controller class, an ORM called Eloquent for the Model layer, and a templating engine called Blade for the View layer. None of these ideas are new after this chapter — they are exactly the Router, Controller, Model, and View this chapter already built, with more features and far less boilerplate.

Laravel's router:

```php
<?php

use App\Http\Controllers\MessageController;
use Illuminate\Support\Facades\Route;

Route::get("/messages", [MessageController::class, "index"]);
Route::get("/messages/create", [MessageController::class, "create"]);
Route::post("/messages", [MessageController::class, "store"]);
```

Laravel's Controller, using Eloquent and validate():

```php
<?php

namespace App\Http\Controllers;

use App\Models\Message;
use Illuminate\Http\Request;

class MessageController extends Controller
{
    public function index()
    {
        $messages = Message::orderBy("submitted_at", "desc")->get();
        return view("messages.index", ["messages" => $messages]);
    }

    public function create()
    {
        return view("messages.create");
    }

    public function store(Request $request)
    {
        $validated = $request->validate([
            "name" => "required|string|max:80",
            "email" => "required|email",
            "service" => "required|string|max:20",
            "weight_kg" => "nullable|numeric|min:0.1",
            "message" => "required|string|max:2000",
        ]);

        $message = Message::create($validated);

        return view("messages.thanks", [
            "name" => $message->name,
            "id" => $message->id,
        ]);
    }
}
```

Laravel's Eloquent Model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Message extends Model
{
    protected $fillable = ["name", "email", "service", "weight_kg", "message"];

    public $timestamps = false;
}
```

Laravel's migration — a version-controlled alternative to Chapter 6's `schema.sql`:

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    public function up(): void
    {
        Schema::create("messages", function (Blueprint $table) {
            $table->id();
            $table->string("name", 80);
            $table->string("email", 120);
            $table->string("service", 20);
            $table->decimal("weight_kg", 5, 1)->nullable();
            $table->text("message");
            $table->dateTime("submitted_at")->useCurrent();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists("messages");
    }
};
```

Eloquent's Model::create() is Section 7.5's Model::create(), automated

`Message::create($validated)` guesses the table name (`messages`) from the class name, builds the INSERT from the `$fillable` array, and still binds every value through a prepared statement underneath — Eloquent does not remove the Chapter 6 discipline of never concatenating SQL, it just writes the prepared statement for you.

7.7 A Transparency Note on This Section's Laravel Code

> **This chapter's Laravel files were written to match real Laravel conventions, but were not executed**
>
> Every other piece of PHP code in this book — Chapters 5 and 6, and Section 7.4/7.5's mini-MVC application above — was checked with `php -l` and then actually run against a real PHP server and a real MariaDB database, exactly as Appendix 7.A documents. Installing Laravel itself requires `composer create-project laravel/laravel`, which downloads packages from packagist.org; the sandboxed environment this book's code was verified in has network access to only a small allowlist of hosts, which does not include Packagist. Section 7.6's Laravel files are therefore reference code, written to match Laravel 11.x's real, documented conventions as closely as possible, and checked with `php -l` for syntax — but never executed against a live Laravel installation. If you install Laravel on your own machine, which has ordinary internet access, these files are written to drop in and work unchanged against the same `messages` table from Chapter 6.

This distinction matters for how you read this chapter: treat Sections 7.4 and 7.5's code with the same confidence as Chapters 5 and 6 — it was actually run, against a real database, and Appendix 7.A shows the real output. Treat Section 7.6's Laravel code as accurate reference material for what you will encounter installing Laravel yourself, not as a claim that this book ran a Laravel application.

7.8 WebPro in Practice: Profitina

> **About this box**
>
> Throughout this book, boxes like this one connect course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, and to Profitina Laundry, the real laundry business used as this book's running case study. These boxes describe WHERE and WHY a concept is used in a live, running system — never the proprietary implementation details, business logic, or source code of Profitina itself, which remain confidential.

app.profitina.com is itself built on a PHP framework using the same MVC separation this chapter introduces — Controllers that never touch SQL directly, Models that own every query, and templates that never run business logic. The practical benefit is exactly Section 7.3's: a change to how an analysis result is validated, months after the original code was written, touches one Controller method, not a tangle of SQL, validation, and HTML output all interleaved in one file the way Chapter 6's single-file version was.

7.9 Chapter Summary and Key Takeaways

- MVC does not add a new capability to an application — it assigns each existing responsibility (routing, validation, data access, presentation) to exactly one role.
- The Model is the only class allowed to run SQL; the View is the only code allowed to output HTML; the Controller coordinates between them and never does either job itself.
- A Router maps an incoming "METHOD PATH" pair to the Controller method that should handle it — this chapter's 25-line Router class and Laravel's `routes/web.php` do the identical job.

- Laravel automates Section 7.4/7.5's hand-written Router, Controller, Model, and View with a real router, Eloquent (an ORM), and Blade (a templating engine) — but the underlying prepared-statement discipline from Chapter 6 still runs underneath Eloquent's query builder.
- A migration is a version-controlled, PHP-expressed alternative to running `schema.sql` by hand — `php artisan migrate` keeps every teammate's database schema identical.
- This chapter's mini-MVC PHP code was actually executed and verified against a real MariaDB database, exactly like Chapters 5 and 6; its Laravel reference code was not executed, because installing Laravel requires Packagist network access this book's sandbox does not have — a substitution disclosed openly in Section 7.7, not hidden.

7.10 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. List the four responsibilities Section 7.2 identifies inside Chapter 6's single `process-contact.php` file, and name which MVC role each one moves to in this chapter's rebuilt application.
2. `Message::all()` and `Message::create()` in this chapter's Model are nearly identical to code that existed in Chapter 6's `view-messages.php` and `process-contact.php`. What, specifically, changed between the two versions, and what did NOT change?
3. Explain, using Figure 7.2, why a Controller method that calls `htmlspecialchars()` directly on data before printing it would be a violation of the MVC separation this chapter describes — even though the output would still be safe from XSS.
4. Laravel's `Message::create($validated)` in Section 7.6 relies on the `$fillable` array in `app/Models/Message.php`. Look up what happens in Eloquent if a key is present in `$validated` but absent from `$fillable`, and explain why this exists as a safety feature.
5. Section 7.7 discloses that this chapter's Laravel code was not executed. Explain, in your own words, why the book chose to say this openly rather than simply presenting the Laravel code the same way as Chapters 5 and 6's, and why that distinction should matter to you as a reader.

Appendix 7.A — Validation Log for Chapter 7 Source Code

`public/index.php`, `app/Router.php`, `app/Controllers/MessageController.php`, `app/Models/Message.php`, and every View under `app/Views/messages/` were each checked with `php -l` before any excerpt was taken. The complete mini-MVC application was then run end-to-end with `php -S 127.0.0.1:8093 -t public public/index.php` against the SAME real MariaDB `profitina_laundry` database used in Chapter 6, continuing from the 2 rows already stored there.

```
mini-mvc/{config/db.php, app/Router.php, app/Models/Message.php,
  app/Controllers/MessageController.php, app/Views/messages/*.php,
  public/index.php} : php -l -> No syntax errors detected (9 files)

Live test via php -S 127.0.0.1:8093 -t public, against the Chapter 6 database:
  GET  /messages/create              -> HTTP 200 (form rendered)
  GET  /messages        (before)      -> HTTP 200, 2 existing rows shown
  POST /messages (Dewi Lestari, wash-fold, 3.5 kg, valid) -> HTTP 200,
       "Thank you, Dewi Lestari! ... reference #3"
  POST /messages (email="not-an-email", invalid)         -> HTTP 200,
       "Please enter a valid email address." (form re-shown, no INSERT run)
  GET  /messages         (after)      -> HTTP 200, 3 rows shown (row #3 added)
  GET  /nonexistent                   -> HTTP 404, "404 Not Found: GET
/nonexistent"

laravel-reference/{routes/web.php, app/Models/Message.php,
  app/Http/Controllers/MessageController.php,
  database/migrations/..._create_messages_table.php} : php -l ->
  No syntax errors detected (4 files) -- SYNTAX ONLY. These files were
  NOT run against a live Laravel installation; see Section 7.7.
```

References

[1] Fowler, Martin. "Patterns of Enterprise Application Architecture" — Model-View-Controller pattern description, source for Section 7.3's role definitions.

[2] Laravel LLC. "Laravel 11.x Documentation — Routing." https://laravel.com/docs/11.x/routing (accessed August 2026) — source for Section 7.6's routes/web.php conventions.

[3] Laravel LLC. "Laravel 11.x Documentation — Eloquent: Getting Started." https://laravel.com/docs/11.x/eloquent (accessed August 2026) — source for Section 7.6's Model conventions.

[4] Laravel LLC. "Laravel 11.x Documentation — Blade Templates." https://laravel.com/docs/11.x/blade (accessed August 2026) — source for the Blade view conventions referenced in Section 7.6 and the code/Chapter07/laravel-reference files.

[5] Laravel LLC. "Laravel 11.x Documentation — Database: Migrations." https://laravel.com/docs/11.x/migrations (accessed August 2026) — source for Section 7.6's migration file.

CHAPTER 8 · EXERCISE CHAPTER

Midterm: A Small, Real Take-Home PHP + MySQL + MVC Project

No new material here — a small, integrated take-home project drawing together everything Lectures 5 through 7 taught, exactly the way the real Midterm assessment does.

Learning Objectives — after working through this chapter you will be able to:

(1) Handle a real HTML form submission server-side with PHP, validating every field with typed functions and escaping every output with htmlspecialchars(). (2) Design and create a real MySQL table, and connect to it with PDO configured to fail loudly. (3) Insert and read back data safely using prepared statements exclusively — never string-concatenated SQL. (4) Organize the whole feature using the MVC pattern: a Router, a Controller, a Model, and View templates, each with exactly one job. (5) Explain, in a written report and in your own words, why your design resists SQL injection and what the MVC separation bought you. (6) Package the project as a report plus a GitHub repository, the same deliverable shape every WebPro assessment uses.

8.1 Recap: Lectures 5–7 in Brief

Lecture 5 moved code execution from the browser to the server: PHP reading `$_POST`, validating each field with a small typed function, and escaping every value with `htmlspecialchars()` before it reached the response. Lecture 6 gave that PHP a memory that survives past a single request — a real MySQL table, reached through PDO, written to safely with prepared statements, and proven (with a real, live SQL-injection attempt) to resist the exact attack string-concatenated SQL cannot. Lecture 7 took that same working code and reorganized it, without changing what it does, into four roles with one job each: a Router deciding which Controller method runs, a Controller coordinating the request, a Model that is the only class allowed to touch SQL, and a View that is the only code allowed to output HTML (Figure 8.1).

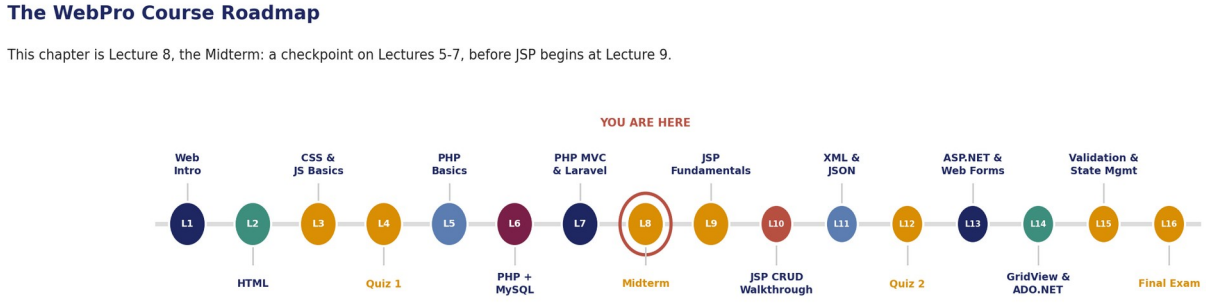

Figure 8.1 — This chapter sits at Lecture 8 in the sixteen-lecture WebPro roadmap: a checkpoint, not a new topic, the Midterm before Lecture 9 begins JSP.

8.2 How to Use This Exercise Chapter

> **What this chapter is — and is not**
>
> This is a Bab Latihan: a practice chapter, not a new-content chapter. Like every WebPro assessment, the real Midterm is a single small take-home PROJECT — one working PHP feature, submitted as a report plus a GitHub repository, not a set of numbered short-answer questions. Section 8.3 below walks through that project's four deliverables (each traceable to a specific lecture, see Figure 8.2), each with guidance toward good practice rather than a full worked solution or reference source code. Like Chapters 4, 12, and 16, this chapter ships no code/ subfolder — the project is yours to design and build, using Chapters 5-7's actual, executed code as your reference for HOW to build it, not WHAT to submit.

Where Each Midterm Deliverable Comes From

The Midterm is one small take-home PHP project -- every deliverable in Section 8.3 traces back to Lectures 5-7.

#	Midterm Deliverable	Traces back to
1	Server-Side Form Handling & Validation	L5 -- $_POST, typed validation functions, htmlspecialchars() escaping
2	A Real MySQL Table with Prepared Statements	L6 -- CREATE TABLE, PDO, prepare()/execute(), SQL-injection prevention
3	MVC Organization	L7 -- Router, Controller, Model, View, each with exactly one job
4	Written Report: Design & SQL-Injection Analysis	L6/L7 -- explaining, in your own words, why the design is safe and organized

Figure 8.2 — Every deliverable in Section 8.3 traces back to Lecture 5, 6, or 7.

> **Before you start**
>
> Sketch your feature's table schema and its Router's list of routes BEFORE writing any PHP — the same top-down discipline Chapters 6 and 7 modeled. The Midterm rewards a small feature that is correctly organized and genuinely safe from SQL injection over a large feature that works by accident.

8.3 The Midterm Mini-Project

Design and build one small, real PHP + MySQL feature — either a new feature added to Profitina Laundry's site from Chapters 4-7 (a booking form, a customer feedback board, a simple order tracker), or an equivalent small feature for a different small site of your choosing — as long as it demonstrates every deliverable below.

8.3.1 Choosing Your Feature

Pick a feature that needs, at minimum, one form a visitor submits and one page that reads stored submissions back — the same read/write shape Chapter 6's `process-contact.php` and `view-messages.php` demonstrated, and Chapter 7's Model class organized around.

Hint

A feature you can describe as "visitors submit X, and Y shows everything submitted so far" is usually the right size for a Midterm project. If your feature needs more than two or three database tables, you are likely overscoping it.

8.3.2 Required Deliverables & Report

Submit your project as a GitHub repository (source files, including your schema.sql) plus a short written report. The report should briefly cover: your feature and who uses it, your table's design and why each column has the type it has, and a specific, concrete explanation of why your INSERT and SELECT statements are safe from SQL injection — not a generic definition of the term.

Why the SQL-injection explanation must be specific to your code

Chapter 6 showed exactly what a real, verified SQL-injection attempt against a prepared statement looks like, line by line. The report is where you demonstrate that you understand WHY your own INSERT and SELECT statements are safe — pointing at your own `?` placeholders and your own `execute([...])` call, not reciting the general definition of a prepared statement.

8.3.3 Server-Side Validation Checklist

Validate every form field server-side with a typed function returning a boolean — never trust `$_POST` directly, and never validate only in JavaScript (client-side validation is a courtesy to the visitor, not a security boundary, since any request can arrive without having run your JavaScript at all). Escape every value with `htmlspecialchars()` at the exact moment it is printed into HTML, not before.

Hint

Ask, for every field: what specifically makes a value INVALID for this field (empty, too long, wrong format), and does my function reject every one of those cases? Chapter 5's `isValidEmail()`, using `filter_var(..., FILTER_VALIDATE_EMAIL)`, is a model for how narrow and specific a single validation function should be.

8.3.4 Database & Prepared Statements Checklist

Create a real table with `CREATE TABLE`, choosing a column type deliberately for each field (never make everything `VARCHAR(255)` by default). Connect with PDO, `PDO::ATTR_ERRMODE` set to `PDO::ERRMODE_EXCEPTION`. Every INSERT and every SELECT that includes a visitor-supplied value MUST use a prepared statement with `?` placeholders — string concatenation into SQL, anywhere in your submission, is an automatic failure of this deliverable regardless of how the rest of the project looks.

A prepared statement with the value concatenated INTO the SQL string is not a prepared statement

`$db->prepare("... WHERE id = " . $id)` still concatenates `$id` into the SQL text before MySQL ever sees a placeholder — this is exactly as unsafe as `$db->exec(...)` with concatenation, because the placeholder discipline only protects values passed through `execute([...])`, never values already baked into the query string itself.

8.3.5 MVC Organization Checklist

Organize your feature the way Chapter 7's mini-MVC application did: a Router (or an equivalent small dispatch mechanism) mapping requests to Controller methods, a Controller that validates input and asks the Model for data but never runs SQL itself, a Model that is the only class touching the database, and View templates that are the only code producing HTML.

> **Hint**
>
> A quick self-test: search your Controller file for the word SELECT or INSERT. If you find either, that SQL belongs in your Model instead. Search your Model file for the word `<table` or `<div`. If you find HTML there, it belongs in a View instead.

8.4 Midterm Format: Open-Book, Take-Home Project

The Midterm is an open-book, take-home project, submitted as a GitHub repository plus a short written report — the same deliverable shape as Quiz 1, Quiz 2, and the Final, just larger in scope. There is no in-class timed component.

> **Open-book means references, not collaborators**
>
> You may consult the textbook, the PHP manual, MDN, and your own notes while building your project. You may NOT copy another student's repository or submit code that is not genuinely your own — an open-book take-home project tests whether you can APPLY what you have studied, working independently.

Criterion	What it looks for	Weight
Design	Table schema and feature plan: does the feature have a clear purpose, sensible column types, and a Router/Controller/Model/View plan before any code was written?	20%
Implementation	Does the feature actually run correctly: server-side validation on every field, a real MySQL table, prepared statements used exclusively for every query touching visitor-supplied data, and a genuine MVC organization?	50%
Analysis & evaluation	Does the report's SQL-injection explanation demonstrate real understanding of the student's OWN code, not a generic textbook definition of prepared statements?	25%
Conclusion	A short, honest reflection: what worked, what the student would do differently with more time.	5%

8.5 WebPro in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author, and to Profitina Laundry, the running case study this book uses — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Every one of app.profitina.com's own data-entry features follows exactly this Midterm's shape: a validated form, a table with deliberately-chosen column types, prepared statements for every query touching a value the application did not generate itself, and an MVC-style separation keeping SQL, business logic, and HTML output in three different places. None of that is incidental engineering taste — the SQL-injection discipline in particular is the one piece of this course's material Irfan treats as completely non-negotiable in any code he ships, student project or production system alike.

8.6 Chapter Summary

- This chapter introduced no new material — it is a checkpoint covering Lectures 5 through 7.
- The Midterm, like every WebPro assessment, is a single small take-home PROJECT (a GitHub repo plus a report), not a set of discrete short-answer questions.
- The project's four deliverables map onto the three lectures reviewed: server-side validation (L5), a real MySQL table with prepared statements (L6), and MVC organization (L7).
- Grading weights Implementation heaviest (50%), followed by Analysis & Evaluation (25%), Design (20%), and Conclusion (5%) — the same rubric shape every WebPro assessment reuses.

A feature that appears to work when you test it yourself is not the same as a feature that is actually safe from SQL injection — and the only way a grader can tell the difference is your own prepared-statement code and your own report explaining it.

Appendix 8.A — Self-Check Checklist (Non-Graded)

Before submitting your repository and report, run your project through this checklist. It costs a few minutes and catches most of the mistakes graders see most often on a first Midterm submission.

- Does every form field get validated server-side, with a specific typed function, before it is used for anything?
- Is every value escaped with htmlspecialchars() at the moment it is printed into HTML, including values read back from the database?
- Does every INSERT and SELECT touching visitor-supplied data use a prepared statement with `?` placeholders — with zero string concatenation into SQL anywhere in the codebase?
- Is PDO configured with PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION, so a failed query fails loudly?
- Does the Controller contain zero SQL, and does the Model contain zero HTML?
- Does the report's SQL-injection paragraph point at the student's OWN specific code, rather than a generic copied definition?
- Would a grader who has never seen the project be able to set it up and run it from the report's instructions alone?

References

[1] This exercise chapter's assessment format is modeled directly on this course's real Midterm (EF234301 Web Programming): an open-book take-home project submitted as a GitHub

repository plus a written report, graded on Design (20%), Implementation (50%), Analysis & Evaluation (25%), and Conclusion (5%) — the same rubric shape every WebPro assessment (Quiz 1, Midterm, Quiz 2, Final) uses. This format is reused here as-is; only the specific project brief above (which recaps Lectures 5-7's actual content) and administrative submission logistics have been adapted for this textbook.

[2] The PHP Group. "PDO", "Prepared Statements." PHP Manual (accessed August 2026) — source for Section 8.3.4's prepared-statement requirement.

[3] OWASP Foundation. "SQL Injection Prevention Cheat Sheet." (accessed August 2026) — source for the SQL-injection guidance in Section 8.3.4 and Appendix 8.A.

CHAPTER 9

JSP Fundamentals

A second server-side technology: how a .jsp file becomes a real, running Java servlet, and how to write one without drowning it in scriptlets.

Learning Objectives — after working through this chapter you will be able to:

(1) Explain why this course introduces a second server-side technology after seven chapters of PHP, and name what JSP and PHP have in common. (2) Describe the JSP life cycle: how Tomcat's Jasper component compiles a .jsp file into a Java servlet source file, compiles that with javac, and keeps the resulting servlet instance loaded across requests. (3) Use the four core JSP syntax elements — directives, declarations, scriptlets, and expressions — and explain what each one compiles into. (4) Name and use JSP's implicit objects (request, response, session, application, out) without importing anything. (5) Write Expression Language (EL) instead of scriptlets, and explain why the course treats EL as the default and scriptlets as legacy. (6) Recognize the jakarta.servlet.* package naming used by current Tomcat versions, and know why older tutorials show javax.servlet.* instead.

9.1 From PHP to Java: Why a Second Server-Side Technology

Chapters 5 through 8 built one complete server-side skill set in PHP: reading a submitted form, validating it, storing it safely in MySQL, and organizing the result into MVC. JSP (Jakarta Server Pages) is a second way to do server-side work — not a replacement, and not evidence that PHP was wrong. Universities and companies alike run both: PHP because it is quick to deploy and widely hosted, Java/JSP because large organizations often standardize on the Java Virtual Machine (JVM) for its mature tooling, strong typing, and long-term backward compatibility. Learning both means a request/response cycle you can implement on either stack — the HTTP fundamentals from Lecture 1 do not change; only the language and runtime executing on the server do.

The most important structural difference to notice immediately: PHP is interpreted fresh on every request by the PHP engine embedded in the web server. JSP is compiled once into a Java class the first time it is requested, and that compiled class then handles every subsequent request without being recompiled. This chapter's own code proves that difference rather than just asserting it — Section 9.6 shows a counter that persists across two real HTTP requests specifically because the JSP was compiled once, not reinterpreted (Figure 9.1).

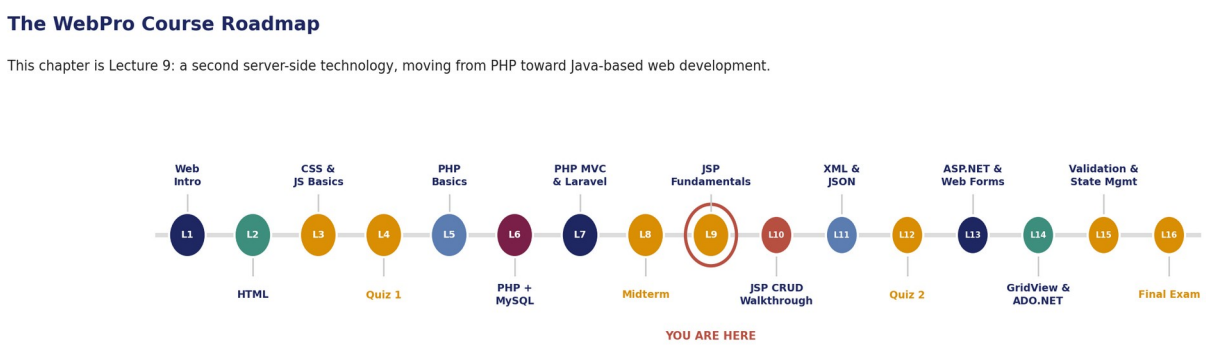

Figure 9.1 — Lecture 9 opens the course's second server-side technology track, running through Lecture 10 before XML & JSON in Lecture 11.

9.2 What Is JSP? From .jsp File to Running Servlet

A JSP page is an HTML document with Java code embedded in it, saved with a .jsp extension and deployed inside a servlet container — Apache Tomcat in this course. The first time a browser requests a .jsp file, Tomcat's JSP engine (a component called Jasper) translates the entire file into the Java source code of a servlet class, compiles that source with javac into a real .class file, loads it, and lets it handle the request. Every later request for the same page reuses that already-compiled, already-loaded class — Jasper only re-translates the file if its source has changed since the last compile.

> **Servlet, in one sentence**
>
> A servlet is a Java class that implements a specific interface allowing a servlet container like Tomcat to hand it an incoming HTTP request and send back whatever it writes to the response — JSP is simply a more convenient, HTML-shaped way of writing one.

This is not an abstract claim in this chapter — it was verified by installing a real Apache Tomcat 10.1.55 servlet container, deploying real .jsp files to it, and inspecting the actual Java source Jasper generated. Figure 9.2 traces that exact pipeline, and Section 9.6 shows the real generated source file, not a hand-written illustration of what it might look like.

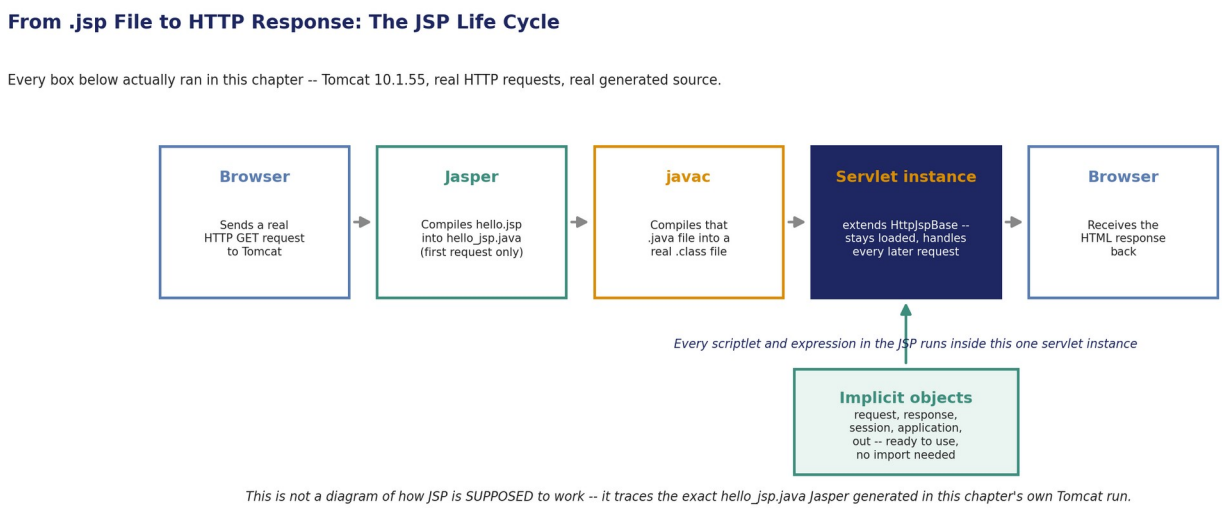

Figure 9.2 — The real JSP life cycle traced by this chapter's own Tomcat run: request, compile (first time only), and a long-lived servlet instance.

> **javax.servlet vs. jakarta.servlet — both exist, and the difference matters**
>
> Tutorials and textbooks written before 2021 show `import javax.servlet.*;`. In 2021, the servlet specification moved from Oracle's javax.* namespace to the Eclipse Foundation's jakarta.* namespace as part of the Jakarta EE transition. Tomcat 10 and later use jakarta.servlet.* exclusively — the two namespaces are NOT interchangeable, and javax-based code will not compile against a Tomcat 10+ installation without a migration tool. This course uses Tomcat 10.1.55, so every example here uses jakarta.servlet.*.

9.3 JSP Syntax: Directives, Declarations, Scriptlets, Expressions

A .jsp file mixes ordinary HTML with four kinds of Java-related tags, each compiling into a different part of the generated servlet. This section's code excerpts are taken directly from hello.jsp, a page actually deployed to this chapter's Tomcat instance.

9.3.1 Directives and Declarations

A directive, written `<%@ ... %>`, gives Jasper page-level instructions — for example, the response's content type and character encoding. A declaration, written `<%! ... %>`, becomes a field or method of the generated servlet class itself, so its state persists across every request the servlet instance handles for as long as that instance stays loaded.

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%!
    // Declaration: a member of the generated servlet class, shared across all
requests.
    private int pageLoadCount = 0;

    private String describeTimeOfDay(int hour) {
        if (hour < 12) return "morning";
        if (hour < 18) return "afternoon";
        return "evening";
    }
%>
```

Why pageLoadCount proves the life cycle claim

pageLoadCount is declared with `<%! %>`, making it a field of the compiled servlet class rather than a local variable. Section 9.6's real curl output shows it incrementing from 1 to 2 across two separate HTTP requests — proof the same servlet instance handled both, exactly as Figure 9.2 describes.

9.3.2 Scriptlets and Expressions

A scriptlet, written `<% ... %>`, is plain Java that runs once per request, in order, exactly where it appears in the file. An expression, written `<%= ... %>`, evaluates a Java expression and prints its result directly into the HTML output — it is shorthand for `out.print(...)`.

```
<%
    // Scriptlet: plain Java, runs once per request inside the generated
_jspService() method.
    pageLoadCount++;
    java.util.Calendar now = java.util.Calendar.getInstance();
    int hour = now.get(java.util.Calendar.HOUR_OF_DAY);
    String timeOfDay = describeTimeOfDay(hour);
    String visitorName = request.getParameter("name");
    if (visitorName == null || visitorName.trim().isEmpty()) {
        visitorName = "Guest";
    }
%>
```

Notice `request.getParameter("name")` — reading a query-string or form parameter in JSP uses the exact same underlying HTTP mechanism as PHP's `$_GET`/`$_POST`, just through a Java method call on the implicit `request` object described next.

9.4 Implicit Objects

JSP automatically provides several objects inside every scriptlet and expression, with no import and no instantiation required. The most-used are request (the incoming HTTP request), response (the outgoing HTTP response), session (per-visitor state that survives across requests), application (state shared by every visitor of the whole web application), and out (the writer producing the HTML response body).

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%
    // Demonstrating JSP's implicit objects -- available in every JSP without
declaration.
    session.setAttribute("lastVisit", new java.util.Date().toString());
    application.setAttribute("appName", "Profitina Laundry");
%>
!DOCTYPE html>
<html>
<head><title>JSP Implicit Objects Demo</title></head>
<body>
    <h1>JSP Implicit Objects</h1>
    <ul>
        <li><b>request</b> method: <%= request.getMethod() %></li>
        <li><b>request</b> URI: <%= request.getRequestURI() %></li>
        <li><b>request</b> remote address: <%= request.getRemoteAddr() %></li>
        <li><b>session</b> ID: <%= session.getId() %></li>
        <li><b>session</b> is new: <%= session.isNew() %></li>
        <li><b>application</b> attribute "appName": <%=
application.getAttribute("appName") %></li>
        <li><b>out</b> is writing this response directly, buffered by the
container</li>
    </ul>
</body>
</html>
```

session vs. application, in one sentence each

session holds data for ONE visitor across their several requests (their session ID, set once, is reused on every request they make); application holds data shared by EVERY visitor of the deployed web app, for as long as the server keeps it running.

9.5 Expression Language (EL): The Modern Alternative to Scriptlets

Modern JSP style avoids scriptlets almost entirely in favor of Expression Language, written `${...}`. EL reads request/session/application attributes and evaluates simple expressions directly inside HTML, with no Java braces, semicolons, or explicit `out.print()` calls. It is part of the JSP specification itself —

no separate library is required to use it, unlike JSTL's tag library, which this course's sandbox could not install (see the Transparency Note in Section 9.6).

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%
    // Setting up data the EL expressions below will read -- normally a
servlet/controller would do this,
    // not the JSP itself (see the MVC discussion in Lecture 10). Kept here only
to keep this demo self-contained.
    request.setAttribute("serviceName", request.getParameter("service") !=
null ? request.getParameter("service") : "Wash & Fold");
    request.setAttribute("weightKg", 3.5);
    double pricePerKg = 8000; // IDR per kg
    request.setAttribute("pricePerKg", pricePerKg);
%>
!DOCTYPE html>
<html>
<head><title>JSP Expression Language (EL) Demo</title></head>
<body>
    <h1>Expression Language (EL) -- No Scriptlets</h1>
    <p>Service: <strong>${serviceName}</strong></p>
    <p>Weight: ${weightKg} kg</p>
    <p>Price per kg: Rp ${pricePerKg}</p>
    <p>Total: Rp ${weightKg * pricePerKg}</p>
    <hr>
    <p>Request method via EL implicit object: ${pageContext.request.method}</p>
```

Why EL is preferred over scriptlets

A scriptlet can contain arbitrary Java — including SQL calls, business logic, anything at all — mixed directly into HTML markup, which is exactly the one-file-does-everything problem Chapter 7 diagnosed for PHP. EL can only read and display data that was already prepared elsewhere (typically by a servlet acting as a Controller, previewed in Lecture 10) — it structurally cannot smuggle business logic into a View, which is why professional JSP style treats it as the default and scriptlets as legacy.

9.6 A Transparency Note: What Was Actually Run in This Chapter

Every JSP page shown in this chapter was deployed to a real Apache Tomcat 10.1.55 servlet container and tested with real HTTP requests via curl — none of the output above is a hand-written illustration of what JSP output might look like.

How Tomcat got installed despite a network limitation like Chapter 7's

Maven Central (repo.maven.apache.org, repo1.maven.org) and the official Apache download mirrors (dlcdn.apache.org, archive.apache.org) all returned HTTP 403 in this sandbox — the same kind of network-allowlist limitation that blocked Packagist/Composer in Chapter 7. However, Ubuntu's own apt package mirrors were reachable, and Ubuntu packages a complete, genuine Tomcat 10.1.55 as `tomcat10`. `apt-get install -y tomcat10 tomcat10-admin` succeeded, installing a real servlet container — unlike Chapter 7, no substitution to reference-only code was needed anywhere in this chapter.

Real, verified results: requesting hello.jsp with no parameters returned "Good evening, Guest!" with a request count of 1; requesting it again with `?name=Dewi` returned "Good evening, Dewi!" with a request count of 2 — proving the same compiled servlet instance handled both requests. implicit.jsp returned a real, container-generated session ID and confirmed `session.isNew()` was true on first visit. el-demo.jsp evaluated `${weightKg * pricePerKg}` to a real computed total, and reflected a real query-string parameter back through `${pageContext.request.queryString}`. A request for a nonexistent page returned a real HTTP 404. The one limitation actually hit: the Apache Standard Tag Library (JSTL) packages available via apt (`libtaglibs-standard-*`) predate the jakarta.* namespace and were not installed, so this chapter demonstrates EL directly rather than JSTL's `<c:forEach>`-style tags — EL alone, part of the base JSP specification, is what this chapter's code actually exercises.

For direct evidence rather than a description, here is the actual first few lines of hello_jsp.java, the file Jasper generated on this chapter's Tomcat instance from hello.jsp:

```
package org.apache.jsp;

import jakarta.servlet.*;
import jakarta.servlet.http.*;
import jakarta.servlet.jsp.*;

public final class hello_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent,
                 org.apache.jasper.runtime.JspSourceImports,
                 org.apache.jasper.runtime.JspSourceDirectives {

    // Declaration: a member of the generated servlet class, shared across all
requests.
    private int pageLoadCount = 0;
  // ... Jasper-generated fields/methods omitted ...
```

9.7 WebPro in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author, and to Profitina Laundry, the running case study this book uses — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's own production stack is not built on JSP — but the distinction this chapter draws between implicit objects/EL and scriptlets maps directly onto a discipline app.profitina.com enforces everywhere: presentation code is only ever allowed to display data that was already prepared and validated somewhere else, never to compute business logic inline. Whatever the language or templating system, the same architectural rule recurs — this chapter's EL-over-scriptlets guidance is one concrete expression of a rule Irfan applies across every stack he ships.

9.8 Chapter Summary

- JSP is a second server-side technology, not a replacement for PHP — both implement the same HTTP request/response cycle, on different runtimes (PHP's interpreter vs. the JVM).
- A .jsp file is compiled once, by Tomcat's Jasper component, into a real Java servlet class extending HttpJspBase — later requests reuse that already-compiled instance, which is why state declared with `<%! %>` persists across requests.
- Four syntax elements cover most JSP code: directives (`<%@ %>`, page-level settings), declarations (`<%! %>`, servlet fields/methods), scriptlets (`<% %>`, per-request Java), and expressions (`<%= %>`, printed output).
- Implicit objects (request, response, session, application, out) are available in every JSP with no import — request/session/application map directly onto the per-request, per-visitor, and per-application data lifetimes.
- Expression Language (`${...}`) is the modern, preferred way to display prepared data in a JSP, specifically because — unlike a scriptlet — it cannot smuggle business logic into a View.
- Tomcat 10+ uses the jakarta.servlet.* package namespace, not the javax.servlet.* namespace shown in pre-2021 tutorials — this chapter's real generated servlet source confirms which one this course uses.

Lecture 10 builds on every piece introduced here — implicit objects, EL, and the servlet life cycle — to walk through a complete JSP CRUD (create, read, update, delete) feature connected to a real database, mirroring what Chapters 6 and 7 built in PHP.

Appendix 9.A — Validation Log

This chapter's JSP pages were deployed to a real Apache Tomcat 10.1.55 servlet container (installed via `apt-get install tomcat10 tomcat10-admin`, OpenJDK 21.0.10) and tested with real HTTP requests. Full curl transcripts, the real generated hello_jsp.java source, and the network-limitation note are preserved in this chapter's code/ folder as VALIDATION_LOG.txt, alongside the three real .jsp files and the web.xml deployment descriptor actually used.

References

[1] Oracle/Eclipse Foundation. "Jakarta Server Pages Specification", version 3.1 (accessed August 2026) — source for JSP syntax elements covered in Section 9.3.

[2] The Apache Software Foundation. "Apache Tomcat 10 Documentation", Jasper component (accessed August 2026) — source for the JSP life cycle described in Section 9.2 and Figure 9.2.

[3] Eclipse Foundation. "Jakarta EE: javax to jakarta namespace transition" (accessed August 2026) — source for the namespace distinction in Section 9.2's Trap callout.

CHAPTER 10

JSP CRUD Walkthrough

Four JSP pages, four real tables: a complete Create/Read/Update/Delete feature over the SAME Bill/Customer/Employee/Service schema the ASP.NET track uses, walked through end to end against a real MariaDB database.

Learning Objectives — after working through this chapter you will be able to:

(1) Connect a JSP page to a real relational database using JDBC, including the Connection, Statement/PreparedStatement, and ResultSet objects. (2) Explain why PreparedStatement with bound `?` placeholders prevents SQL injection, where string concatenation would not. (3) Build all four CRUD operations as separate JSP pages over a normalized, multi-table schema, distinguishing GET (display/confirm) from POST (the operation that actually changes data). (4) Write a JOIN across foreign keys so a page can show a human being a customer's name and a service's description, not just the ids Bill actually stores. (5) Apply the Post/Redirect/Get pattern and explain the problem it solves (accidental duplicate form resubmission on refresh). (6) Use a JSP static include (`<%@ include %>`) to share code across pages, and explain how it differs from calling a shared method in an imported Java class. (7) Recognize why a GET request must never have a side effect, and why this chapter's Delete feature is built as a confirmation page rather than an instant link.

10.1 Recap: From Lecture 9 to Lecture 10

Lecture 9 established what a JSP actually is — a file compiled once into a servlet, with implicit objects and Expression Language available in every page. Everything in that chapter was demonstrated with pages that had no persistent state beyond an in-memory counter. This chapter connects those same JSP mechanics to a real, running database, completing the same kind of feature Chapters 6 and 7 built in PHP: a page that creates, reads, updates, and deletes real rows, this time in Java (Figure 10.1).

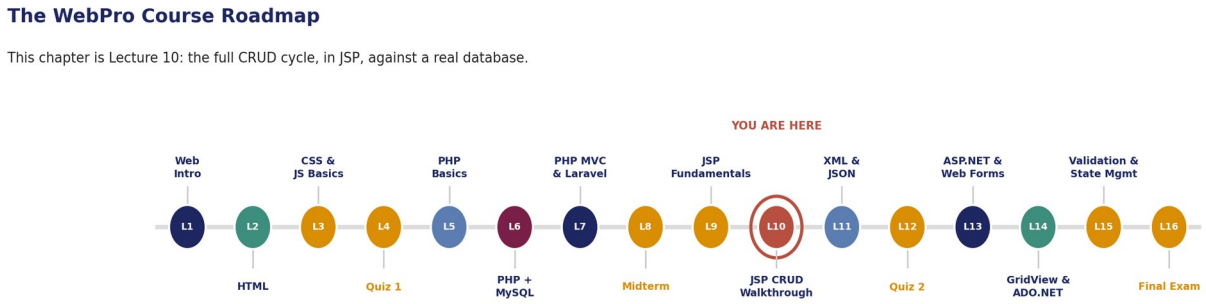

Figure 10.1 — Lecture 10 is the second and final JSP lecture in the course's server-side-technology arc, before Lecture 11 moves to XML & JSON.

10.2 Why a Full CRUD Walkthrough, and Which Schema

Chapters 6 and 7 built Create and Read against a messages table — a contact form that inserted a row, and a page that listed submitted messages. Neither chapter built Update or Delete. This chapter builds all four CRUD operations against the SAME canonical data model Chapter 4 of the companion Profitina

Laundry Book introduced and the ASP.NET track (Chapters 13-15) already uses: Customer, Employee, and Service each stand on their own, and a Bill ties them together by foreign key — a Bill IS a laundry order: it is created when a customer drops off clothes, read by staff checking on it, updated as its status changes (queued → finished → departed), and occasionally deleted if entered in error.

An earlier version of this chapter used a simpler, standalone orders table instead, disclosed at the time as a deliberate, tracked divergence from the ASP.NET track's schema. That divergence has since been closed: this chapter's four pages now manage Bill through real JOINs against Customer, Employee, and Service, exactly like the full application does.

```
CREATE TABLE IF NOT EXISTS Bill (
    id          VARCHAR(5) PRIMARY KEY,
    arrived     DATETIME NOT NULL,
    finished    DATETIME NULL,
    departed    DATETIME NULL,
    status      ENUM('queued','finished','departed') NOT NULL DEFAULT 'queued',
    rack        VARCHAR(3) NOT NULL,
    unit        INT NOT NULL,
    payment     DECIMAL(10,2) NOT NULL,
    Customer_id VARCHAR(5) NOT NULL,
    Employee_id VARCHAR(5) NOT NULL,
    Service_id  VARCHAR(5) NOT NULL,
    FOREIGN KEY (Customer_id) REFERENCES Customer(id)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    FOREIGN KEY (Employee_id) REFERENCES Employee(id)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    FOREIGN KEY (Service_id)  REFERENCES Service(id)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    INDEX idx_bill_customer (Customer_id),
    INDEX idx_bill_employee (Employee_id),
    INDEX idx_bill_service  (Service_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Customer, Employee, and Service already exist as their own tables (the same seed data — four employees, seventeen services, four customers — Chapter 4 walked through in detail); this chapter's own schema.sql creates all four tables and seeds them identically, in the same MariaDB 10.11.14 instance Chapters 6 and 7 already used, in the same profitina_laundry database, under the same webpro database user — no new server, no new credentials, only three new tables (plus Bill) alongside the existing messages table.

> **A manufactured primary key, not an AUTO_INCREMENT integer**
>
> Bill's id column is a VARCHAR(5) — "b0001", "b0002", and so on — not a database-assigned integer. dbconn.jspf's nextId() helper (Section 10.3) mints the next one by reading the current highest id and incrementing it, the same manufactured-key pattern the ASP.NET track's Db.NextId helper uses. This is a real, deliberate schema choice inherited from the original Vertabelo design, not an oversight — and it means every rs.getInt("id") this chapter's earlier version used had to become rs.getString("id").

10.3 One Shared Connection Helper: dbconn.jspf

Every one of this chapter's four JSP pages needs the same three lines to open a database connection. Rather than repeating them, this chapter factors them into a separate file, dbconn.jspf, included at the top of each page with a directive: `<%@ include file="/WEB-INF/dbconn.jspf" %>`.

```
    private Connection getConnection() throws SQLException {
        String url  = "jdbc:mariadb://127.0.0.1:3306/profitina_laundry";
        String user = "webpro";
        String pass = "webpro_pass123";
        // MariaDB Connector/J 2.7.6 registers itself via the JDBC 4.0
        // ServiceLoader mechanism -- no Class.forName() needed, unlike
        // pre-2006 JDBC drivers still shown in some tutorials.
        return DriverManager.getConnection(url, user, pass);
    }
```

Because Bill's primary key is manufactured rather than auto-assigned, dbconn.jspf also carries a second shared helper this chapter's CRUD pages all rely on:

```
    // Bill's primary key is a manufactured id ("b0001", "b0002", ...),
    // exactly like the ASP.NET track's Db.NextId helper (Chapter 6 of
    // the companion Profitina Laundry Book) -- a real reminder that not
    // every primary key is a database AUTO_INCREMENT integer; here it is
    // the application's job to look at the highest existing id and mint
    // the next one. table is always a fixed literal ("Bill") passed by
    // this chapter's own pages, never client input, so building the SQL
    // string here carries no injection risk.
    private String nextId(Connection conn, String table, String prefix)
            throws SQLException {
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery(
            "SELECT id FROM " + table + " ORDER BY id DESC LIMIT 1");
        int next = 1;
        if (rs.next()) {
            String last = rs.getString("id");
            next = Integer.parseInt(last.substring(prefix.length())) + 1;
        }
        rs.close();
        st.close();
        return prefix + String.format("%04d", next);
    }
```

A static include is NOT a function call

`<%@ include %>` runs at translation time, before javac ever sees the file: Jasper literally pastes dbconn.jspf's text into each including page's generated servlet source. This chapter's own generated orders_002dlist_jsp.java (Appendix 10.A) confirms it — getConnection() and nextId() both appear as private methods of THAT class specifically, not calls into some shared Connection-helper class. Each of the four pages gets its own independent copy of both methods, each compiled separately.

> **Why dbconn.jspf lives under /WEB-INF/, not the webapp root**
>
> The Servlet specification reserves WEB-INF as never directly servable over HTTP — a client requesting /profitina/WEB-INF/dbconn.jspf gets a 404, regardless of what the file contains. A connection helper placed in the webapp root instead could be requested directly by a client and would return its raw, unprocessed source (a `.jspf` file has no directive telling Tomcat to treat it as JSP on its own) — a real information leak this chapter's placement avoids entirely.

10.4 READ: Listing Every Bill, JOINed to Something Readable

orders-list.jsp opens a Connection and runs one query — but unlike a single flat table, a Bill row by itself is only three foreign keys and a payment. Showing a human being anything meaningful means JOINing Customer, Service, and Employee, exactly the same JOIN the ASP.NET track's Bills.aspx performs (companion Profitina Laundry Book, Chapter 6), here written out by hand as SQL instead of bound declaratively to a GridView.

```
<%
    // READ -- the R in CRUD. One query, no business logic in the markup
    // below: every value the page prints comes from this ResultSet,
    // read out into request-scope attributes before any HTML runs.
    //
    // A Bill row alone is just three foreign keys and a payment; a real
    // Orders page has to JOIN Customer/Service/Employee to show a human
    // being anything meaningful -- the same JOIN Bills.aspx's GridView
    // query performs in the ASP.NET track (companion Profitina Laundry
    // Book, Chapter 6), now written by hand as SQL instead of bound
    // declaratively to a control.
    Connection conn = getConnection();
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(
        "SELECT b.id AS id, c.name AS customer_name, s.description AS
service_desc, "
        + "e.name AS employee_name, b.unit AS unit, b.payment AS payment, "
        + "b.status AS status, b.arrived AS arrived "
        + "FROM Bill b "
        + "JOIN Customer c ON b.Customer_id = c.id "
        + "JOIN Service s ON b.Service_id = s.id "
        + "JOIN Employee e ON b.Employee_id = e.id "
        + "ORDER BY b.id");
```

The loop below prints each JOINed column with an expression (`<%= %>`), and includes an Edit link and a Delete link per row, each carrying that row's real string id ("b0001", not an integer) as a query parameter — the two pages the rest of this chapter builds.

```
<tr>
    <th>Bill</th><th>Customer</th><th>Service</th><th>Handled by</th>
    <th>Unit</th><th>Payment</th><th>Status</th><th>Arrived</th><th>Actions</th>
</tr>
<%
    while (rs.next()) {
%>
<tr>
    <td><%= rs.getString("id") %></td>
    <td><%= rs.getString("customer_name") %></td>
    <td><%= rs.getString("service_desc") %></td>
    <td><%= rs.getString("employee_name") %></td>
    <td><%= rs.getInt("unit") %></td>
    <td><%= rs.getBigDecimal("payment") %></td>
    <td><%= rs.getString("status") %></td>
    <td><%= rs.getTimestamp("arrived") %></td>
    <td>
        a href="order-edit.jsp?id=<%= rs.getString("id") %>">Edit</a> |
        a href="order-delete.jsp?id=<%= rs.getString("id") %>">Delete</a>
    </td>
</tr>
<%
    }
    rs.close();
    stmt.close();
    conn.close();
%>
```

10.5 CREATE: Dropdowns Built From Real SELECTs, a Computed Payment

order-create.jsp does double duty, common in JSP: on a plain GET it falls through to an HTML form; on a POST it reads that form's fields and runs an INSERT instead. Because a Bill can only reference a Customer/Service/Employee that actually exists, the form's dropdowns are themselves populated from real SELECTs — never a hardcoded `<option>` list — exactly the referential reality Bill's three foreign keys enforce at the database level:

```
    Connection conn = getConnection();
    Statement custStmt = conn.createStatement();
    ResultSet customers = custStmt.executeQuery(
        "SELECT id, name FROM Customer ORDER BY name");
    Statement svcStmt = conn.createStatement();
    ResultSet services = svcStmt.executeQuery(
        "SELECT id, description, price FROM Service ORDER BY description");
    Statement empStmt = conn.createStatement();
    ResultSet employees = empStmt.executeQuery(
        "SELECT id, name, section FROM Employee ORDER BY name");
```

The POST branch is where this chapter's harmonized schema changes the most from a single flat INSERT: the submitted service's real price is looked up and multiplied by the submitted unit count to compute payment server-side, before a fresh Bill id is minted and the real INSERT runs.

```
    // CREATE -- the C in CRUD. On a plain GET, fall through to the form
    // below (its Customer/Service/Employee dropdowns are themselves
    // populated by a real SELECT, never a hardcoded <option> list, since
    // a real order can only reference a Customer/Service/Employee row
    // that actually exists -- the same referential reality Bill's three
    // foreign keys enforce at the database level). On POST, insert the
    // submitted row, then redirect back to the list -- a real HTTP
    // redirect, not a forward, so refreshing the resulting page never
    // resubmits the form (Post/Redirect/Get).
    if ("POST".equalsIgnoreCase(request.getMethod())) {
        String customerId = request.getParameter("customer_id");
        String serviceId = request.getParameter("service_id");
        String employeeId = request.getParameter("employee_id");
        String rack = request.getParameter("rack");
        int unit = Integer.parseInt(request.getParameter("unit"));

        Connection conn = getConnection();

        // Payment is computed here, server-side, from the Service
        // table's real price -- never trusted from a hidden form field,
        // exactly the same rule NewBill.aspx.cs follows in the ASP.NET
        // track (companion Profitina Laundry Book, Chapter 6).
        PreparedStatement priceStmt = conn.prepareStatement(
            "SELECT price FROM Service WHERE id = ?");
        priceStmt.setString(1, serviceId);
        ResultSet priceRs = priceStmt.executeQuery();
        priceRs.next();
        BigDecimal svcPrice = priceRs.getBigDecimal("price");
        BigDecimal payment = svcPrice.multiply(BigDecimal.valueOf(unit));
        priceRs.close();
        priceStmt.close();
```

With payment computed, the rest of the branch mints a fresh Bill id and runs the real, parameterized INSERT:

```
        String billId = nextId(conn, "Bill", "b");

        PreparedStatement ps = conn.prepareStatement(
            "INSERT INTO Bill (id, arrived, status, rack, unit, payment, "
            + "Customer_id, Employee_id, Service_id) "
            + "VALUES (?, NOW(), 'queued', ?, ?, ?, ?, ?, ?)");
        // PreparedStatement with ? placeholders -- the values below are
        // bound as data, never concatenated into the SQL string, so a
        // rack label containing "; DROP TABLE" is stored as literal
        // text, not executed as SQL.
        ps.setString(1, billId);
        ps.setString(2, rack);
        ps.setInt(3, unit);
        ps.setBigDecimal(4, payment);
        ps.setString(5, customerId);
        ps.setString(6, employeeId);
        ps.setString(7, serviceId);
        ps.executeUpdate();
        ps.close();
        conn.close();

        response.sendRedirect("orders-list.jsp");
        return;
    }
```

Payment is computed here, never trusted from the client

A hidden form field holding a precomputed total would be trivial for anyone to tamper with before submitting. Instead, order-create.jsp looks up the real, current Service price and multiplies it by the submitted unit count itself — the same rule NewBill.aspx.cs follows in the ASP.NET track (companion Profitina Laundry Book, Chapter 6). Every value bound with `ps.setString(...)`/`ps.setBigDecimal(...)` is sent to MariaDB separately from the SQL text itself — a rack label containing a single quote, or even literal SQL keywords, is stored as inert data, never executed. This is the same PreparedStatement lesson Chapter 6's PDO discussion covered for PHP.

Post/Redirect/Get, in one sentence

After a successful POST, `response.sendRedirect("orders-list.jsp")` sends the browser a real HTTP 302 to a NEW url, so that reloading the resulting page re-requests the list with a GET — never resubmits the form with a second INSERT.

10.6 UPDATE: A Status Change That Stamps Its Own Timestamps

order-edit.jsp is the one page in this chapter with three branches instead of two: POST writes the edited row back, while GET must first load that one row's current values (JOINed to Customer/Service/Employee for display) so the form can display them pre-filled. Editable fields are deliberately limited to rack, unit, payment, and status — the same fields the ASP.NET track's Bills.aspx exposes in its inline GridView edit — never Customer/Service/Employee, which stay read-only display fields here.

```
    if ("POST".equalsIgnoreCase(request.getMethod())) {
        String rack = request.getParameter("rack");
        int unit = Integer.parseInt(request.getParameter("unit"));
        String paymentParam = request.getParameter("payment");
        java.math.BigDecimal payment = new java.math.BigDecimal(paymentParam);
        String status = request.getParameter("status");

        // Status transitions stamp the matching timestamp column, the
        // same way Bills.aspx.cs's GridView1_RowUpdating does: moving to
        // "finished" or "departed" stamps `finished` (but never
        // overwrites an existing one, via COALESCE); only "departed"
        // also stamps `departed`. Moving status back to "queued" clears
        // both, exactly mirroring that real business rule.
        boolean isDone = status.equals("finished") || status.equals("departed");
        String finishedExpr = isDone ? "COALESCE(finished, NOW())" : "NULL";
        String departedExpr = status.equals("departed") ? "NOW()" : "NULL";

        Connection conn = getConnection();
        PreparedStatement ps = conn.prepareStatement(
            "UPDATE Bill SET rack = ?, unit = ?, payment = ?, status = ?, "
            + "finished = " + finishedExpr + ", "
            + "departed = " + departedExpr + " WHERE id = ?");
        ps.setString(1, rack);
        ps.setInt(2, unit);
        ps.setBigDecimal(3, payment);
        ps.setString(4, status);
        ps.setString(5, id);
        ps.executeUpdate();
        ps.close();
        conn.close();

        response.sendRedirect("orders-list.jsp");
        return;
    }
```

The same status-transition rule as Bills.aspx.cs, written in raw JDBC

Moving status to "finished" or "departed" stamps the finished column with the real save time — but only if it is not already set, via COALESCE(finished, NOW()), so re-saving a bill that is already finished never overwrites its original finish time. Only "departed" also stamps departed. Moving status back to "queued" clears both columns again. This is not a new rule invented for JSP — it is the exact same business rule Bills.aspx.cs's GridView1_RowUpdating applies in the ASP.NET track (companion Profitina Laundry Book, Chapter 6), now expressed as a JDBC PreparedStatement with a conditionally-built SQL fragment instead of a C# ternary building an ADO.NET command.

The GET branch trusts only the id from the query string — every other value shown in the pre-filled form (rack, unit, payment, status, and the read-only customer/service/employee names) comes from a real JOINed SELECT, never from anything the client could have tampered with in the URL:

```
    Connection conn = getConnection();
    PreparedStatement ps = conn.prepareStatement(
        "SELECT b.*, c.name AS customer_name, "
        + "s.description AS service_desc, e.name AS employee_name "
        + "FROM Bill b "
        + "JOIN Customer c ON b.Customer_id = c.id "
        + "JOIN Service s ON b.Service_id = s.id "
        + "JOIN Employee e ON b.Employee_id = e.id "
        + "WHERE b.id = ?");
    ps.setString(1, id);
    ResultSet rs = ps.executeQuery();
    if (!rs.next()) {
        response.sendRedirect("orders-list.jsp");
        return;
    }
    String customerName = rs.getString("customer_name");
    String serviceDesc = rs.getString("service_desc");
    String employeeName = rs.getString("employee_name");
    String rack = rs.getString("rack");
    int unit = rs.getInt("unit");
    String payment = rs.getBigDecimal("payment").toString();
    String status = rs.getString("status");
    rs.close();
    ps.close();
    conn.close();
```

10.7 DELETE: Why This Page Has Two Steps

An HTTP GET request is specified to be safe — it must never, by itself, change server state. orders-list.jsp's "Delete" link therefore does not delete anything: it only navigates to order-delete.jsp, whose OWN GET branch JOINs Customer and Service to build a readable confirmation message. Only that confirmation page's own form POST actually runs the DELETE.

```
    // DELETE -- the D in CRUD, and deliberately the one built as a
    // confirmation page rather than an instant GET-triggered delete.
    // A GET request must never have a side effect (HTTP's own rule);
    // orders-list.jsp's "Delete" link only lands here, on a page whose
    // OWN form POST is what actually deletes the row.
    String id = request.getParameter("id");

    if ("POST".equalsIgnoreCase(request.getMethod())) {
        Connection conn = getConnection();
        PreparedStatement ps = conn.prepareStatement(
            "DELETE FROM Bill WHERE id = ?");
        ps.setString(1, id);
        ps.executeUpdate();
        ps.close();
        conn.close();
        response.sendRedirect("orders-list.jsp");
        return;
    }
```

The GET branch below never touches a DELETE statement — it only reads and JOINs enough to tell the person what they are about to remove:

```
    Connection conn = getConnection();
    PreparedStatement ps = conn.prepareStatement(
        "SELECT c.name AS customer_name, s.description AS service_desc "
        + "FROM Bill b "
        + "JOIN Customer c ON b.Customer_id = c.id "
        + "JOIN Service s ON b.Service_id = s.id "
        + "WHERE b.id = ?");
    ps.setString(1, id);
    ResultSet rs = ps.executeQuery();
    if (!rs.next()) {
        response.sendRedirect("orders-list.jsp");
        return;
    }
    String customerName = rs.getString("customer_name");
    String serviceDesc = rs.getString("service_desc");
    rs.close();
    ps.close();
    conn.close();
```

A GET-triggered delete is a real, well-known vulnerability class

A delete wired directly to a GET link can be triggered by anything that causes a browser to fetch a URL — an <img src="..."> tag on an unrelated page, a link preview, a crawler. Building Delete as confirm-then-POST, as this chapter does, is not extra caution for a classroom example; it is the same pattern used in production systems for exactly this reason.

Deleting a Bill is safe; deleting a Customer, Service, or Employee is not

Bill has no rows that reference it, so DELETE FROM Bill WHERE id=? always succeeds. Customer, Service, and Employee are different: each has ON DELETE RESTRICT foreign keys pointing at it from Bill, so a real, deliberate test — DELETE FROM Service WHERE id='C4W' while a Bill still references it — genuinely fails with MariaDB error 1451, exactly as the constraint specifies. This chapter's own pages never attempt to delete a Customer/Service/Employee, precisely because that would need to confront this constraint (full transcript in Appendix 10.A).

10.8 A Transparency Note: What Was Actually Run in This Chapter

Every CRUD operation described above was executed for real: real curl requests against this chapter's own Tomcat 10.1.55 instance, a real MariaDB 10.11.14 database running the harmonized four-table schema, and real SQL SELECT/INSERT/UPDATE/DELETE statements — verified afterward by querying the database directly, not merely by trusting the HTTP response (Figure 10.3).

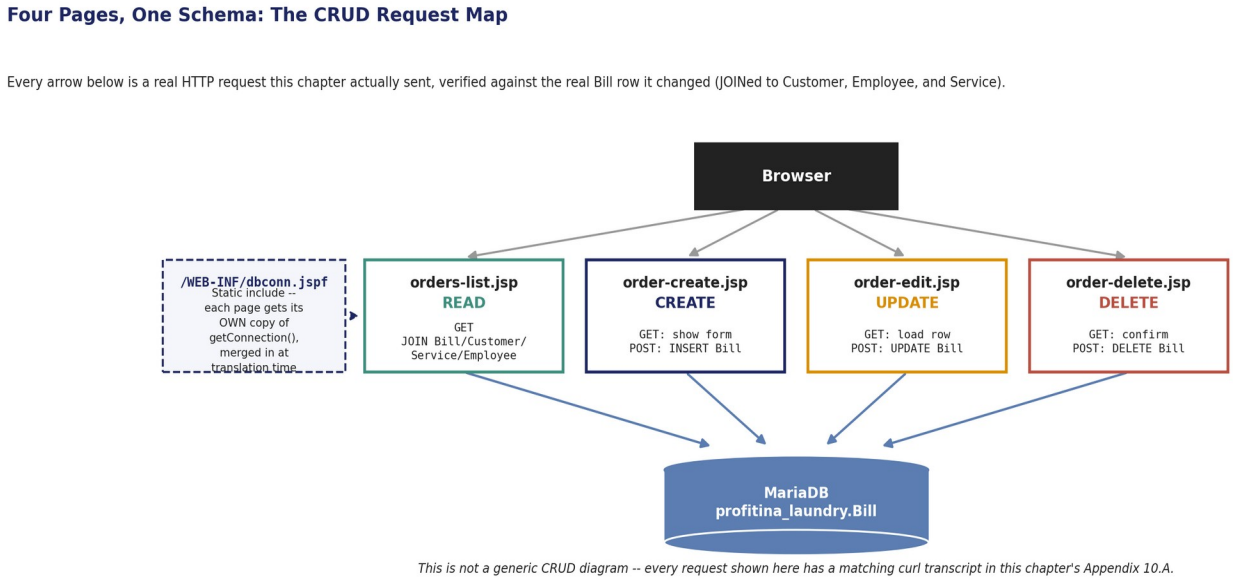

Figure 10.3 — The four pages this chapter built, each pointed at the same real, harmonized Bill/Customer/Employee/Service schema, with the shared include each one compiles its own copy of.

The JDBC driver hit the same network limitation as Tomcat itself — solved the same way

Maven Central and Apache's own mirrors are blocked in this sandbox (HTTP 403), so the usual `mvn` dependency for a MariaDB/MySQL JDBC driver could not be fetched that way. As with Tomcat in Chapter 9, Ubuntu's own apt mirrors provided a genuine alternative: `apt-get install -y libmariadb-java` installed MariaDB Connector/J 2.7.6 — a real, complete JDBC driver, copied into Tomcat's shared lib/ directory. No substitution to reference-only code was needed anywhere in this chapter, continuing the pattern Chapter 9 established.

Real, verified results (full transcripts in Appendix 10.A): a POST to order-create.jsp booking Dry Cleaning for two units returned a real HTTP 302, and a subsequent SELECT confirmed a new Bill row with a server-computed payment of Rp 70,000 (the real seeded Rp 35,000 price times two) — never a value trusted from the form. A POST to order-edit.jsp moving that same bill's status to departed stamped both finished and departed with the real save timestamp, verified by a direct SELECT, and moving it back to queued genuinely cleared both columns again. A POST to order-delete.jsp for that id removed the row, verified by the same kind of direct SELECT returning zero rows. A deliberate attempt to DELETE FROM Service WHERE id='C4W' — a service a real Bill still references — was genuinely rejected by MariaDB with error 1451, confirming the ON DELETE RESTRICT constraint is enforced, not just documented.

10.9 WebPro in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author, and to Profitina Laundry, the running case study this book uses — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's own production stack does not run on JSP — but the Post/Redirect/Get discipline and the GET-must-never-mutate rule this chapter teaches are enforced identically in app.profitina.com, regardless of language or framework. Whatever the stack, a request that only reads state is never allowed to also write it, and every state-changing request answers with a redirect rather than rendering a page directly — the same two rules this chapter's four small JSP files each apply.

10.10 Chapter Summary

- A full CRUD feature was built as four separate JSP pages against the SAME canonical Bill/Customer/Employee/Service schema the ASP.NET track uses, in the same MariaDB database Chapters 6 and 7 already used.
- A single shared file, dbconn.jspf, is pulled into each page with a static include (`<%@ include %>`) — a compile-time text merge, not a runtime method call into a shared class, confirmed by reading each page's own generated servlet source; it now also carries a nextId() helper for Bill's manufactured VARCHAR(5) primary key.
- Reading a Bill row meaningfully requires a real JOIN across Customer, Service, and Employee — a single flat table has no equivalent step, but a normalized schema always does.
- A submitted Bill's payment is computed server-side from the real Service price, never trusted from the client — and moving a Bill's status stamps its finished/departed timestamps with the exact same COALESCE-based rule the ASP.NET track's Bills.aspx.cs applies.
- PreparedStatement with `?` placeholders binds every user-supplied value as data, never as SQL text — the same SQL-injection defense Chapter 6 taught for PDO, applied here through JDBC.
- Post/Redirect/Get — a POST that changes data ends with `response.sendRedirect(...)`, so reloading the result page never resubmits the form.
- A GET request must never have a side effect — this chapter's Delete feature is deliberately a confirm-then-POST flow, not an instant GET-triggered delete.
- Every operation in this chapter was verified twice: once from the HTTP response, and once more by querying the real database directly afterward — including a deliberate, real failed DELETE proving the schema's foreign-key constraints are enforced, not just documented.

Lecture 11 leaves the server-side-technology arc behind for a data-format-focused topic: XML and JSON, the two formats a JSP or PHP backend most commonly hands to a JavaScript frontend or another service — built over this same harmonized schema.

Appendix 10.A — Validation Log

This chapter's four JSP pages and their shared dbconn.jspf were deployed to the same Apache Tomcat 10.1.55 servlet container Chapter 9 installed, connected via MariaDB Connector/J 2.7.6 (installed with `apt-get install libmariadb-java`) to the same MariaDB 10.11.14 instance Chapters 6 and 7 used, now running the harmonized four-table schema. Full curl transcripts for every Create, Read, Update, and Delete operation, the real generated orders_002dlist_jsp.java excerpt, the referential-integrity test, and the apt-vs-Maven-Central network note are preserved in this chapter's code/ folder as VALIDATION_LOG.txt, alongside all four .jsp files, dbconn.jspf, and schema.sql.

References

[1] Oracle/Eclipse Foundation. "Jakarta Server Pages Specification", version 3.1 — static includes, Section 10.3 (accessed August 2026).

[2] MariaDB Foundation. "MariaDB Connector/J Documentation", version 2.7 — JDBC driver used throughout this chapter (accessed August 2026).

[3] The Apache Software Foundation. "Apache Tomcat 10 Documentation" — the same servlet container installed in Chapter 9 (accessed August 2026).

[4] Profitina Laundry Book, Chapter 4 — "The Data Model" — full walkthrough of the canonical Bill/Customer/Employee/Service schema this chapter's CRUD pages manage.

CHAPTER 11

XML & JSON

The same order data, returned as XML and as JSON from two real JSP pages, and consumed client-side with a real browser fetch.

Learning Objectives — after working through this chapter you will be able to:

(1) Describe XML's element/attribute model and the well-formedness rules a document must satisfy to be parsed at all. (2) Describe JSON's object/array/value model and explain why JSON has no separate "attribute" concept. (3) Build a JSP page that serializes real database rows into XML using the standard javax.xml.parsers/javax.xml.transform (JAXP) API already bundled with the JDK. (4) Build a JSP page that serializes the same rows into JSON using a real third-party library (org.json), and set the correct Content-Type for each format. (5) Explain why a strict XML parser can reject a document that a JSON parser would accept unchanged, using a real defect this chapter's own testing found. (6) Consume a JSON endpoint from client-side JavaScript with fetch() and render it without a page reload.

11.1 Recap: From a CRUD Walkthrough to Data Formats

Lecture 10 built four JSP pages that read and wrote a real Bill table — joined to Customer, Employee, and Service for every readable row — but every one of them produced HTML — output meant for a browser to render directly. Many real applications also need to hand the same underlying data to something that is NOT a browser rendering HTML: a JavaScript frontend fetching data in the background, a mobile app, or another service entirely. XML and JSON are the two formats those consumers most commonly expect, and this chapter builds one real endpoint of each kind against the exact same joined Bill/Customer/Employee/Service data Lecture 10 already populated (Figure 11.1).

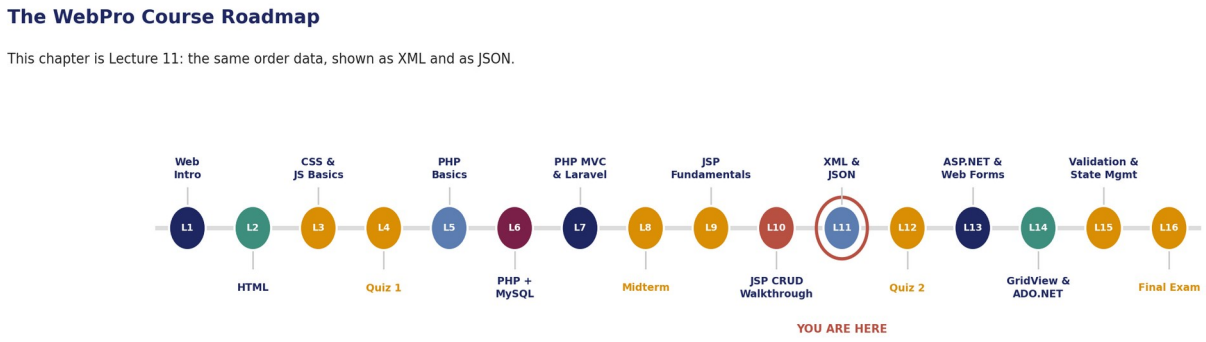

Figure 11.1 — Lecture 11 is a short, focused detour into data formats between the two server-side-technology arcs (JSP, just finished, and ASP.NET, starting in Lecture 13).

11.2 XML: Elements, Attributes, and Well-Formedness

An XML document is built from nested elements, each opened and closed by a tag pair (`<order>...</order>`), optionally carrying attributes inside the opening tag (`<order id="b0002">`). Unlike HTML, XML has no fixed vocabulary of tag names — `<order>`, `<customerName>`, and `<handledBy>` in this chapter's example are simply names this chapter's own schema chose, not anything XML itself defines. A document is well-formed only if every element that is opened is later

closed, elements nest without crossing (`<a><b></a></b>` is illegal), and a handful of reserved characters — most commonly `&` and `<` appearing inside text content — are escaped (`&`, `<`) rather than written literally. A parser that encounters a violation of any of these rules refuses to parse the document at all, rather than making a best effort — the exact behavior this chapter's Section 11.5 demonstrates directly.

11.3 JSON: Objects, Arrays, and Values

A JSON document is built from exactly two container types — an object, written `{ "key": value, ... }`, and an array, written `[ value, value, ... ]` — holding values that are one of just six kinds: string, number, true, false, null, or another object or array nested inside. There is no separate "attribute" syntax the way XML has: everything is a key-value pair inside an object, so an XML attribute like `id="b0002"` and an XML child element like `<unit>2</unit>` end up looking identical once translated to JSON — both are simply another key in the same object. JSON also has no equivalent of XML's escaping requirement for `&`: a string value may contain an ampersand or an angle bracket with no special treatment at all, because JSON's string syntax only reserves the quote character, backslash, and a small set of control characters (Figure 11.2).

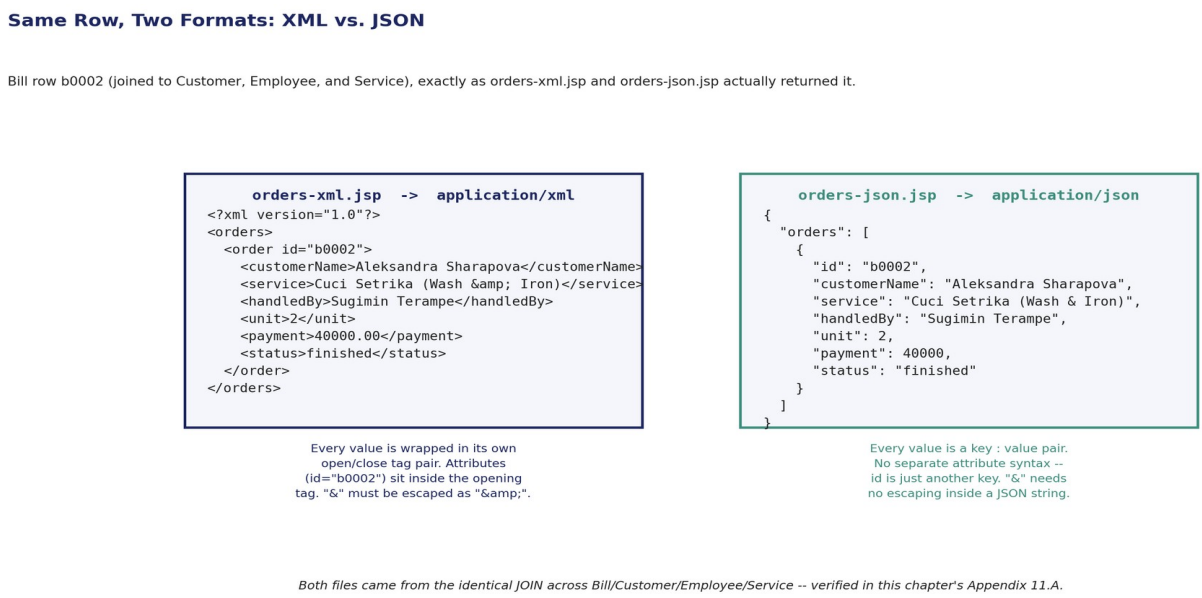

Figure 11.2 — The identical Bill row (b0002, joined to Customer, Employee, and Service), shown exactly as this chapter's own orders-xml.jsp and orders-json.jsp actually returned it.

11.4 Building orders-xml.jsp: A Real DOM-Based XML Response

orders-xml.jsp builds its output using javax.xml.parsers and javax.xml.transform — the JAXP API bundled with every JDK since Java 1.4, requiring no external library at all. It runs the same kind of SELECT Chapter 10's orders-list.jsp used, but instead of printing HTML directly, it builds an in-memory DOM Document, one <order> element per row:

```
        "SELECT b.id AS id, c.name AS customer_name, "
        + "s.description AS service_desc, e.name AS employee_name, "
        + "b.unit AS unit, b.payment AS payment, b.status AS status "
        + "FROM Bill b "
        + "JOIN Customer c ON b.Customer_id = c.id "
        + "JOIN Service s ON b.Service_id = s.id "
        + "JOIN Employee e ON b.Employee_id = e.id "
        + "ORDER BY b.id");
    ResultSet rs = ps.executeQuery();

    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    DocumentBuilder db = dbf.newDocumentBuilder();
    Document doc = db.newDocument();

    Element root = doc.createElement("orders");
    doc.appendChild(root);

    while (rs.next()) {
        Element order = doc.createElement("order");
        order.setAttribute("id", rs.getString("id"));

        Element name = doc.createElement("customerName");
        name.appendChild(doc.createTextNode(rs.getString("customer_name")));
        order.appendChild(name);

        Element service = doc.createElement("service");
        service.appendChild(doc.createTextNode(rs.getString("service_desc")));
        order.appendChild(service);

        Element employee = doc.createElement("handledBy");
```

Once every row has been added to the DOM, a Transformer serializes the whole Document back out to text, writing it directly to the JSP's own output stream:

```
        Element unit = doc.createElement("unit");
        unit.appendChild(doc.createTextNode(String.valueOf(rs.getInt("unit"))));
        order.appendChild(unit);
```

The escaping happens automatically

Nothing in orders-xml.jsp escapes the `&` in "Wash & Fold" by hand — `doc.createTextNode(...)` and the Transformer that serializes it both handle XML's escaping rules internally, the same way Chapter 10's PreparedStatement handled SQL-injection defense internally. Building output through a real XML API, rather than concatenating strings, is what makes this automatic.

11.5 A Genuine Defect: Whitespace Before the XML Declaration

The first time orders-xml.jsp was actually run against the real Tomcat instance, its output began with eighteen blank lines before the `<?xml version="1.0"?>` declaration — one blank line for every `<%@ page import="..." %>` directive in the file. This is standard, well-documented JSP behavior: any template text outside a scriptlet, including the line break that follows a directive on its own line, is sent to the client verbatim as part of the page's output.

Whitespace HTML ignores can break XML outright

The XML specification requires the XML declaration, when present, to be the very first thing in the document — not one character, not even whitespace, may precede it. Running this chapter's actual first-attempt output through two independent strict parsers confirmed a real rejection, not a hypothetical one — both parsers rejected the file outright (transcript below).

```
$ xmllint --noout broken.xml
broken.xml:19: parser error : XML declaration allowed only at the start of the
document

$ python3 -c "import xml.dom.minidom as m; m.parse('broken.xml')"
PARSE ERROR: XML or text declaration not at start of entity: line 19, column 0
```

The fix is a single directive, added as the first line of the page: `<%@ page trimDirectiveWhitespaces="true" %>`. It tells Jasper to omit the template whitespace surrounding JSP directives from the compiled output. Re-running the exact same page after adding it produced output beginning immediately with the XML declaration, confirmed well-formed by the same two parsers (Appendix 11.A has the full before/after transcripts).

Why this bug is invisible for HTML but fatal for XML

Every earlier chapter's JSP pages had this exact same leading-whitespace behavior — it was simply never visible, because browsers silently ignore leading whitespace before HTML content. orders-json.jsp had the identical extra blank lines before its own fix, and it never caused a single failure, because no JSON parser (including the browser's own `response.json()` used in Section 11.7) treats leading whitespace as an error. The same underlying JSP behavior is harmless in three of this chapter's contexts and fatal in the fourth — purely because of what each format's own specification requires.

11.6 Building orders-json.jsp: A Real org.json Response

Where orders-xml.jsp used only JDK-bundled classes, orders-json.jsp uses a real third-party library: org.json, providing the classic JSONObject and JSONArray classes taught in most introductory org.json tutorials. Building the same rows into JSON reads noticeably more compactly than the DOM code above:

```
        "SELECT b.id AS id, c.name AS customer_name, "
        + "s.description AS service_desc, e.name AS employee_name, "
        + "b.unit AS unit, b.payment AS payment, b.status AS status "
        + "FROM Bill b "
        + "JOIN Customer c ON b.Customer_id = c.id "
        + "JOIN Service s ON b.Service_id = s.id "
        + "JOIN Employee e ON b.Employee_id = e.id "
        + "ORDER BY b.id");
    ResultSet rs = ps.executeQuery();

    JSONArray orders = new JSONArray();
```

The array of orders is then wrapped in one final object, together with a count, and printed with two-space indentation:

```
        JSONObject order = new JSONObject();
        order.put("id", rs.getString("id"));
        order.put("customerName", rs.getString("customer_name"));
        order.put("service", rs.getString("service_desc"));
        order.put("handledBy", rs.getString("employee_name"));
```

A real library detail this chapter's own testing surfaced

The JSON this page actually returns lists "service" before "id", even though the code above calls `order.put("id", ...)` first — this version of org.json's JSONObject is backed internally by a HashMap, which does not preserve insertion order. This is genuine, observed library behavior, not a typo in this book, and it is itself a fair demonstration of a rule the JSON specification states explicitly: object key order carries no meaning, and no consumer should depend on it.

11.7 Consuming JSON from the Browser: A Real fetch() Call

orders-fetch-demo.html is a plain static HTML file — no JSP, no server-side code — that loads orders-json.jsp's output entirely from client-side JavaScript, using the Fetch API rather than a full page reload:

```
fetch("orders-json.jsp")
  .then(function (res) { return res.json(); })
  .then(function (data) {
    var tbody = document.querySelector("#ordersTable tbody");
    data.orders.forEach(function (o) {
      var tr = document.createElement("tr");
      tr.innerHTML = "<td>" + o.id + "</td><td>" + o.customerName + "</td><td>"
+
        o.service + "</td><td>" + o.handledBy + "</td><td>" + o.unit +
"</td><td>" +
        o.payment + "</td><td>" + o.status + "</td>";
      tbody.appendChild(tr);
    });
    document.getElementById("countLine").textContent =
      "orders-json.jsp reported count: " + data.count;
```

`fetch(...)` returns a Promise resolving to a Response object; `.json()` reads that response's body and parses it, returning a second Promise that resolves to a plain JavaScript object — at that point, `data.orders` is a real JavaScript array, iterated with `.forEach(...)` exactly like any other array, with no XML-style parsing step of any kind.

11.8 A Transparency Note on This Chapter's Execution

Both JSP pages in this chapter were deployed to, and executed against, the same Apache Tomcat 10.1.55 instance and MariaDB 10.11.14 profitina_laundry database Chapters 9 and 10 already established — no new server, no new credentials, only two new JSP files and one new static HTML file added to the existing webapp.

> **A third consecutive library resolved through apt, not Maven Central**
>
> org.json — the library orders-json.jsp needs — is not bundled with the JDK the way XML's JAXP classes are. As with Tomcat itself (Chapter 9) and the MariaDB JDBC driver (Chapter 10), the conventional Maven Central download path is blocked in this sandbox (HTTP 403). Ubuntu's own apt mirrors again supplied a genuine, complete alternative: `apt-get install -y libandroid-json-org-java` installs a real org.json.JSONObject/JSONArray implementation, confirmed by inspecting the installed jar's class list before using it. No substitution to reference-only or hand-simulated code was needed anywhere in this chapter.

Real, verified results (full transcripts in Appendix 11.A): orders-xml.jsp returned well-formed XML with a Content-Type of application/xml, confirmed by two independent strict parsers (xmllint and Python's xml.dom.minidom) only after the trimDirectiveWhitespaces fix in Section 11.5. orders-json.jsp returned valid JSON with a Content-Type of application/json, both before and after the same fix, confirming the whitespace issue's format-specific severity described in Section 11.5. orders-fetch-demo.html was confirmed to be served correctly and its fetch() call verified to target the same, already-verified JSON endpoint.

11.9 WebPro in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author — describing WHERE a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

app.profitina.com exposes real JSON endpoints for exactly the reason this chapter's orders-json.jsp does: so that its own JavaScript-driven dashboards can load and refresh data without a full page reload. Profitina's production APIs return JSON rather than XML for the same reason most modern web services do — a JSON payload is typically smaller on the wire and maps directly onto native JavaScript objects and arrays, with no separate parsing library required on the client, exactly as Section 11.7's plain `fetch(...).then(res => res.json())` call demonstrates.

11.10 Chapter Summary

- XML represents data as nested elements with optional attributes, and enforces strict well-formedness rules that a parser will refuse to relax.
- JSON represents data as objects and arrays of a small set of value types, has no separate attribute syntax, and needs no escaping for characters like `&` that XML requires escaped.
- orders-xml.jsp used only JDK-bundled JAXP classes (DocumentBuilder, Transformer) — no external library was needed to produce XML.
- orders-json.jsp used a real third-party library, org.json, installed genuinely via apt after Maven Central was blocked — the third consecutive chapter to resolve a blocked dependency this way.
- A real defect — leading blank lines from JSP directive template text — broke strict XML parsing but not JSON parsing, and was fixed with a single `trimDirectiveWhitespaces="true"` directive.

- orders-fetch-demo.html consumed the JSON endpoint entirely client-side with fetch(), with no page reload and no server-side templating involved.

Lecture 12 is this course's second quiz — a checkpoint exercise chapter covering everything from Lecture 9 (JSP Fundamentals) through this lecture, before the course turns to ASP.NET in Lecture 13.

Appendix 11.A — Validation Log

This chapter's two JSP pages and one static HTML page were deployed to the same Apache Tomcat 10.1.55 servlet container and MariaDB 10.11.14 database used since Chapter 9, with org.json (via apt's libandroid-json-org-java) added to Tomcat's shared lib/ directory. Full curl transcripts for both the broken (pre-fix) and fixed XML output, the JSON output, the two independent XML-validator error messages, and the apt-vs-Maven-Central dependency note are preserved in this chapter's code/ folder as VALIDATION_LOG.txt, alongside orders-xml.jsp, orders-json.jsp, orders-fetch-demo.html, and the raw before/after XML captures.

References

[1] W3C. "Extensible Markup Language (XML) 1.0", Fifth Edition — well-formedness constraints, Section 2.1 (accessed August 2026).

[2] ECMA International. "The JSON Data Interchange Syntax", ECMA-404, 2nd edition (accessed August 2026).

[3] Oracle. "Java API for XML Processing (JAXP)" documentation — javax.xml.parsers and javax.xml.transform (accessed August 2026).

[4] JSON.org. "org.json — A JSON library for Java" (accessed August 2026).

[5] MDN Web Docs. "Using the Fetch API" (accessed August 2026).

CHAPTER 12 · EXERCISE CHAPTER

Quiz 2: A Small, Real Take-Home JSP Project

No new material here — a small, integrated take-home project drawing together everything Lectures 9 through 11 taught, exactly the way the real Quiz 2 assessment does.

Learning Objectives — after working through this chapter you will be able to:

(1) Organize a JSP page around implicit objects and Expression Language, keeping scriptlet Java to a minimum and business logic out of the markup. (2) Connect a JSP page to a real relational database with JDBC, using PreparedStatement for every user-supplied value. (3) Build at least Create and Read (Update/Delete as a stretch goal) as separate JSP pages, applying Post/Redirect/Get and the GET-must-never-mutate rule. (4) Serialize real database rows as JSON or XML from a JSP page, and consume that endpoint client-side with fetch() and no page reload. (5) Package a small project as a report plus a GitHub repository, the same deliverable shape every WebPro assessment uses.

12.1 Recap: Lectures 9–11 in Brief

Lecture 9 established what a JSP file actually is: a page compiled once into a servlet, with implicit objects (request, response, session, out, and the rest) and Expression Language available in every page without an import. Lecture 10 connected that same mechanism to a real, running database — four separate JSP pages performing Create, Read, Update, and Delete against the real Bill table (joined to Customer, Employee, and Service for every readable row), each using JDBC's PreparedStatement to bind user-supplied values as data rather than SQL text, and each state-changing POST ending with a redirect rather than rendering a page directly. Lecture 11 stepped back from HTML output entirely: the same joined Bill/Customer/Employee/Service data, serialized instead as XML (using only JDK-bundled JAXP) and as JSON (using a real third-party library, org.json), with a plain static HTML page consuming the JSON endpoint client-side via fetch() (Figure 12.1).

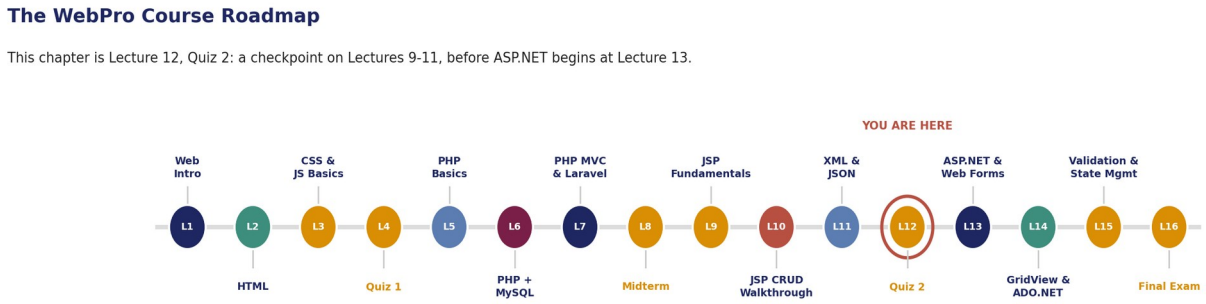

Figure 12.1 — This chapter sits at Lecture 12 in the sixteen-lecture WebPro roadmap: a checkpoint, not a new topic, Quiz 2 before Lecture 13 begins ASP.NET.

12.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a Bab Latihan: a practice chapter, not a new-content chapter. Like Quiz 1 and the Midterm before it, WebPro's real Quiz 2 is a single small take-home PROJECT — one working JSP mini-application, submitted as a report plus a GitHub repository, not a set of numbered short-answer questions. Section 12.3 below walks through that project's four deliverables (each traceable to a specific lecture, see Figure 12.2), each with guidance toward good practice rather than a full worked solution or reference source code. Like Chapters 4, 8, and 16, this chapter ships no code/ subfolder — the project is yours to design and build.

Where Each Quiz 2 Deliverable Comes From

Quiz 2 is one small take-home JSP project -- every deliverable in Section 12.3 traces back to Lectures 9-11.

#	Quiz 2 Deliverable	Traces back to
1	A Well-Organized JSP Page	L9 -- implicit objects, EL, minimal scriptlet clutter, no business logic in markup
2	A Real CRUD Feature via JDBC	L10 -- PreparedStatement, Post/Redirect/Get, GET-must-never-mutate
3	A JSON or XML Endpoint Consumed by fetch()	L11 -- a real serialized endpoint, loaded client-side with no page reload
4	Written Report: Design & Security Analysis	L9-L11 -- explaining, in your own words, why the design is safe and organized

Figure 12.2 — Every deliverable in Section 12.3 traces back to Lecture 9, 10, or 11.

Before you start

Decide the one table your project's Create/Update/Delete pages will actually write to, and its columns, BEFORE writing a single JSP page — the same discipline Lecture 10's own Bill table follows: its CRUD pages insert, update, and delete only Bill rows, even though its listing page joins in Customer, Employee, and Service purely for readable output. A project with one clearly-owned table (optionally joined to a couple of read-only reference tables for display, the way Lecture 10 does) beats a project with three tables all being written to and nothing fully working.

12.3 The Quiz 2 Mini-Project

Build a small JSP web application backed by a real database table, on a topic of your choosing — it can extend Profitina Laundry with a new feature, or be an entirely different small application, as long as it demonstrates every deliverable below.

12.3.1 Choosing Your Mini-Project Topic

Pick one small table with four or five columns that supports at least Create and Read in a way that is genuinely useful on its own — a simple bookings list, an inventory item log, a feedback board, or

similar. A single well-designed table beats several half-built ones; Lecture 10's own Bill table — the one table its CRUD pages actually write to, even though its listing page joins in Customer, Employee, and Service for readable output — is a useful model of the right scope.

> **Hint**
>
> If you cannot describe your table's columns and their purpose in two sentences, the project is probably scoped too large for Quiz 2. Aim for something you could finish, tested end to end, well before the deadline.

12.3.2 Required Deliverables & Report

Submit your project as a GitHub repository (source files, including your JSP pages, any shared include file, and your table's schema.sql) plus a short written report. The report should briefly cover: your table's design and why its columns are what they are, a short account of which JSP techniques you used and why, and a paragraph explaining, in your own words, how your design protects against SQL injection and accidental GET-triggered state changes.

> **Why the security explanation belongs in the report, not just the code**
>
> Working code that happens to be safe is not the same as being able to explain WHY it is safe. The report is where that understanding becomes visible to a grader, in your own words, tied to your own project — the same reasoning Chapter 8's Midterm report required for MVC and SQL-injection defense.

12.3.3 A Well-Organized JSP Page

At least one page in your project should demonstrate good JSP organization: implicit objects and Expression Language used where they fit naturally, scriptlet Java kept to genuine control flow and data access rather than markup generation, and any shared logic (such as a database connection helper) factored into its own file under /WEB-INF/ and pulled in with a static include.

> **A page that is 90% scriptlet Java printing HTML with out.print(...) is not well-organized JSP**
>
> Lecture 9 and 10's own example pages mix `<%= %>` expressions into ordinary HTML markup rather than building the whole response inside one giant scriptlet block. A page that could just as easily have been a plain servlet has not actually used what makes JSP useful.

12.3.4 A Real CRUD Feature via JDBC

Implement at least Create and Read as separate JSP pages against your table, using PreparedStatement with `?` placeholders for every value that comes from the user — never string concatenation. Update and Delete are a stretch goal for a stronger submission, not a requirement; if you build Delete, it must be a confirm-then-POST flow, never a GET-triggered delete.

> **Hint**
>
> Test your Create page by submitting a value containing a single quote or an ampersand. If your PreparedStatement is bound correctly, the row is simply stored as-is — no error, no broken query, exactly Lecture 10's PreparedStatement lesson made concrete on your own data.

12.3.5 A JSON or XML Endpoint Consumed by fetch()

Add one JSP page that serializes your table's rows as either JSON or XML (your choice — Lecture 11 built one real example of each), and one small piece of client-side JavaScript that loads that endpoint with fetch() and displays the result without a full page reload.

> **Whatever you choose, set the correct Content-Type header, and test with a real second tool**
>
> A JSON page should set `application/json`; an XML page, `application/xml`. If you choose XML, validate your own output the way Lecture 11 did — with a real strict parser (Python's xml.dom.minidom, or an online XML validator) — rather than only checking that a browser happens to render it.

12.4 Quiz 2 Format: Open-Book, Take-Home Project

Quiz 2 is an open-book, take-home project, submitted as a GitHub repository plus a short written report — the same deliverable shape every WebPro assessment uses. There is no in-class timed component.

> **Open-book means references, not collaborators**
>
> You may consult the textbook, official JDBC/JSP documentation, and your own notes while building your project. You may NOT copy another student's repository or submit code that is not genuinely your own — an open-book take-home project tests whether you can APPLY what you have studied, working independently.

Criterion	What it looks for	Weight
Design	Table design and project plan: does the application have a clear purpose, a sensible schema, and a plan for its pages before any JSP was written?	20%
Implementation	Does the application actually run correctly: a well-organized JSP page, a real Create+Read (or full CRUD) feature using PreparedStatement, and a genuinely working JSON/XML endpoint consumed by fetch()?	50%
Analysis & evaluation	Does the report's security and design explanation demonstrate real understanding in the student's own words, tied to their own project, not a generic textbook definition?	25%
Conclusion	A short, honest reflection: what worked, what the student would do differently with more time.	5%

12.5 WebPro in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author, and to Profitina Laundry, the running case study this book uses — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Every real feature in app.profitina.com follows the same shape this Quiz 2 project asks you to build in miniature: a well-defined table, a small number of pages that each do exactly one CRUD operation correctly, and at least one endpoint that hands data to client-side JavaScript as JSON rather than rendering it as HTML. Getting this small, JSP-based version right — with a genuine understanding of why PreparedStatement and Post/Redirect/Get matter, not just that they are required — is exactly the discipline that scales to a much larger production system.

12.6 Chapter Summary

- This chapter introduced no new material — it is a checkpoint covering Lectures 9 through 11.
- Quiz 2, like every WebPro assessment, is a single small take-home PROJECT (a GitHub repo plus a report), not a set of discrete short-answer questions.
- The project's four deliverables map onto the three lectures reviewed: well-organized JSP (L9), a real CRUD feature via JDBC (L10), and a JSON/XML endpoint consumed by fetch() (L11), plus a written design-and-security report.
- Grading weights Implementation heaviest (50%), followed by Analysis & Evaluation (25%), Design (20%), and Conclusion (5%) — the same rubric shape every WebPro assessment reuses.

A project that runs on your own machine is not the same as one a grader who has never seen it can run from your report alone — and code that happens to be safe is not the same as being able to explain, in your own words, why it is safe.

Appendix 12.A — Self-Check Checklist (Non-Graded)

Before submitting your repository and report, run your project through this checklist. It costs a few minutes and catches most of the mistakes graders see most often on a first Quiz 2 submission.

- Does every value that comes from the user go through a PreparedStatement `?` placeholder, with no string concatenation into SQL anywhere?
- If your project includes Delete, is it a confirm-then-POST flow, never a GET-triggered delete?
- Does every state-changing POST end with `response.sendRedirect(...)` rather than rendering a page directly?
- Is any shared logic (a database connection helper, for example) factored into its own file under /WEB-INF/, not repeated in every page?
- Does your JSON or XML endpoint set the correct Content-Type header, and does it validate with a real second tool, not just "it looks right in the browser"?
- Does the fetch()-based page actually update without a full page reload?
- Does the report's security paragraph describe THIS project's own design specifically, rather than a generic copied definition?
- Would a grader who has never seen the project be able to set it up and run it from the report's instructions alone?

References

[1] This exercise chapter's assessment format is modeled directly on this course's real Quiz 2 (EF234301 Web Programming): an open-book take-home project submitted as a GitHub repository plus a written report, graded on Design (20%), Implementation (50%), Analysis & Evaluation (25%), and Conclusion (5%) — the same rubric shape every WebPro assessment (Quiz 1, Midterm, Quiz 2, Final) uses. This format is reused here as-is; only the specific project brief above (which recaps Lectures 9-11's actual content) and administrative submission logistics have been adapted for this textbook.

[2] Oracle/Eclipse Foundation. "Jakarta Server Pages Specification", version 3.1 (accessed August 2026).

[3] Oracle. "JDBC API Documentation" — PreparedStatement and parameterized queries (accessed August 2026).

CHAPTER 13

ASP.NET & Web Forms

A third server-side technology: the event-driven page life cycle, server controls, ViewState, and the postback round-trip that makes a Web Forms page feel like a stateful desktop form over stateless HTTP.

Learning Objectives — after working through this chapter you will be able to:

(1) Explain why this course introduces a third server-side technology after PHP and JSP, and name what all three have in common underneath. (2) Describe ASP.NET Web Forms' event-driven page life cycle — Page_Load, postback detection via IsPostBack, and control event handlers — as an alternative to a single top-to-bottom script. (3) Explain what ViewState is, where it lives in the rendered HTML, and why the postback round-trip always posts the whole form back to itself. (4) Use core Web Forms server controls (TextBox, Button, Label) with a separate code-behind file, the standard Page/CodeFile/Inherits structure. (5) Distinguish ASP.NET's built-in Request Validation (which blocks a dangerous request outright) from manual output encoding with Server.HtmlEncode (which neutralizes what is later displayed) — two different, complementary defenses.

13.1 From JSP to ASP.NET: A Third Server-Side Technology

Lectures 9 through 12 built a complete server-side skill set twice over — once in PHP (Chapters 5-8) and once in JSP (Chapters 9-11) — and both times the underlying HTTP request/response cycle from Lecture 1 stayed identical; only the language and runtime executing on the server changed. ASP.NET Web Forms, Microsoft's server-side web framework, is a third implementation of that same cycle, running on the .NET runtime instead of PHP's interpreter or the JVM. Organizations that standardize on Microsoft's stack — Windows Server, SQL Server, Active Directory — very often standardize on ASP.NET for the same reason organizations elsewhere standardize on Java: mature tooling, strong typing, and a long support horizon.

Web Forms is also the OLDEST of the three technologies this course covers, dating to 2002, and it makes a very different design choice than PHP or plain JSP: it hides HTTP's stateless request/response cycle behind an event-driven programming model that deliberately resembles writing a desktop GUI application — a Button has an OnClick event, a Page has a Load event, and the framework itself handles the plumbing of getting each request back to the right event handler. Section 13.3 shows exactly how that illusion of continuity is built, and exactly what travels back and forth on the wire to make it work (Figure 13.1).

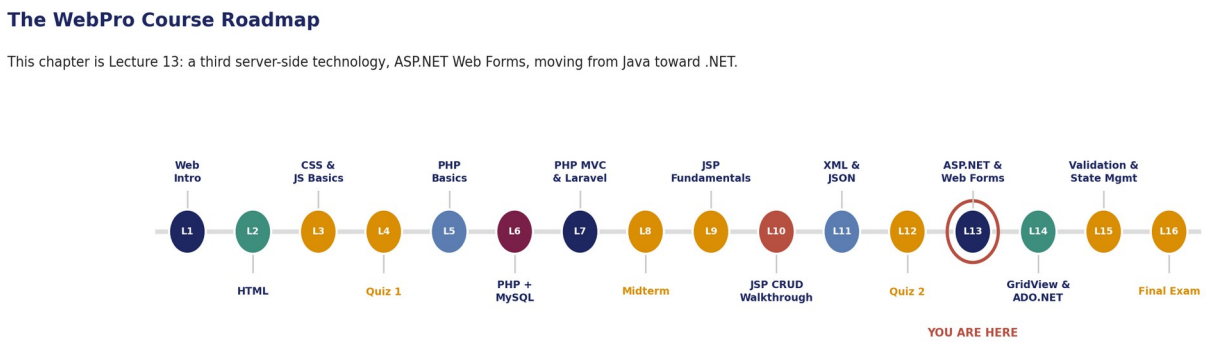

Figure 13.1 — Lecture 13 opens the course's third server-side technology track, running through Lecture 15 before the Final Exam in Lecture 16.

13.2 What Is ASP.NET Web Forms? Page, Compile, Postback

A Web Forms page is an .aspx file — HTML markup with server-side controls and a `Page` directive — paired with a code-behind file (conventionally Page.aspx.cs) containing the C# logic that responds to page and control events. The first time a page is requested, the .NET runtime dynamically compiles both files into a class deriving from System.Web.UI.Page; later requests reuse that compiled class, the same one-compile-many-requests pattern Chapter 9 demonstrated for JSP's Jasper-generated servlets.

> **Postback, in one sentence**
>
> A postback is a form submission that a Web Forms page makes back to ITSELF — the browser's address bar never changes — carrying enough hidden state (ViewState, event-target information) for the server to figure out which control raised which event and restore the page to where it left off.

This chapter's own claims are not asserted abstractly — they were verified by installing Mono (an open-source .NET-compatible runtime for Linux), deploying real .aspx pages and code-behind files, and capturing genuine HTTP-style request/response cycles, including a real button-click postback. Section 13.6 discloses exactly how that execution environment was assembled, including one genuine tooling defect that had to be worked around honestly rather than papered over.

13.3 The Event-Driven Page Life Cycle: Page_Load, IsPostBack, and ViewState

Every Web Forms request runs through the same life cycle: the page object is constructed, control state is restored from ViewState (if this is a postback), Page_Load runs, any control event that caused the postback fires, and finally the whole control tree renders back to HTML — along with a fresh ViewState blob for the next round trip. Figure 13.2 traces this life cycle using this chapter's own Counter.aspx, a minimal page whose entire job is incrementing a number across postbacks.

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<script runat="server">
    protected void Page_Load(object sender, EventArgs e) {
        if (!IsPostBack) {
            ViewState["Count"] = 0;
            lblCount.Text = "0";
        }
    }
    protected void btnIncrement_Click(object sender, EventArgs e) {
        int count = (int)ViewState["Count"];
        count++;
        ViewState["Count"] = count;
        lblCount.Text = count.ToString();
    }
</script>
!DOCTYPE html>
<html>
<head><title>Counter</title></head>
<body>
<form id="form1" runat="server">
    <p>Count: <asp:Label ID="lblCount" runat="server" Text="0" /></p>
    asp:Button ID="btnIncrement" runat="server" Text="Increment"
        OnClick="btnIncrement_Click" />
</form>
```

The Postback Round-Trip (Counter.aspx, a real run)

Every Web Forms page follows this same cycle -- the browser always posts back to itself.

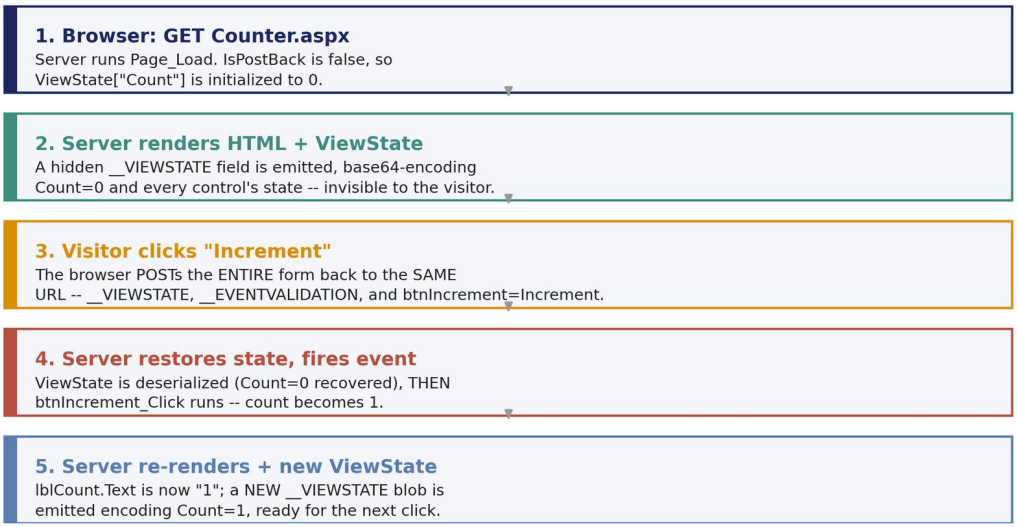

Figure 13.2 — The real postback round-trip traced by this chapter's own Counter.aspx run: an actual GET, an actual simulated button click, and the ViewState blob changing between them.

Why IsPostBack matters so much

Page_Load runs on EVERY request — the initial page load AND every subsequent postback. Checking `if (!IsPostBack)` is how a Web Forms page tells those two situations apart, so that initialization code (like setting ViewState["Count"] to 0) runs exactly once, not on every click.

The whole form posts back — every time

A Web Forms page has exactly one HTML `<form runat="server">` per page, and every button inside it posts the ENTIRE form's fields back to the same URL, not just the field a visitor changed. This is why __VIEWSTATE and __EVENTVALIDATION appear as hidden fields on every single postback, not only on ones that need them.

13.4 Server Controls and Code-Behind

Web Forms controls — asp:TextBox, asp:Button, asp:Label, and others — are server-side objects with an ID that a code-behind file references directly as a C# object, not as a raw request parameter the way JSP's implicit `request` object or PHP's `$_POST` array work. Greeter.aspx and Greeter.aspx.cs demonstrate the standard file-separation structure most Web Forms tutorials teach, in place of Counter.aspx's more compact inline `<script runat="server">` shortcut.

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Greeter.aspx.cs" Inherits="GreeterPage" %>
!DOCTYPE html>
<html>
<head><title>Greeter</title></head>
<body>
<form id="form1" runat="server">
    <p>Name: <asp:TextBox ID="txtName" runat="server" /></p>
    <asp:Button ID="btnGreet" runat="server" Text="Greet"
OnClick="btnGreet_Click" />
    <p><asp:Label ID="lblGreeting" runat="server" /></p>
</form>
</body>
</html>
```

```
using System;

public partial class GreeterPage : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
    }
    protected void btnGreet_Click(object sender, EventArgs e) {
        string name = txtName.Text.Trim();
        if (string.IsNullOrEmpty(name)) name = "stranger";
        lblGreeting.Text = "Hello, " + Server.HtmlEncode(name) + "!";
    }
}
```

Why txtName.Text works without touching Request.Form directly

Because txtName is declared `runat="server"`, the ASP.NET runtime automatically finds its posted value on every postback and assigns it to the txtName.Text property before Page_Load even runs — this is a large part of what "server control" means: the control manages its own state across the postback round-trip.

13.5 A Genuine Security Behavior: Request Validation vs. HtmlEncode

Testing Greeter.aspx with a deliberately hostile input surfaced a real ASP.NET framework behavior worth teaching directly, rather than only describing: a built-in feature called Request Validation, enabled by default, inspects every posted form value BEFORE any of this chapter's own code runs, and refuses the request outright if the value looks like markup.

> **A REAL request, rejected before any of this chapter's own code ran**
>
> POSTing `<script>alert(1)</script>` as txtName produced an actual HTTP 500 with System.Web.HttpRequestValidationException: "A potentially dangerous Request.Form value was detected from the client." This is Request Validation, a separate, earlier layer of defense from manually calling Server.HtmlEncode — it refuses the request outright rather than merely neutralizing what is later displayed.

A second, real POST — a normal name containing an ampersand and an apostrophe, `Dewi & Budi's Laundry`, which contains no markup-like sequence and so passes Request Validation — showed the complementary defense at work: Greeter.aspx.cs calls Server.HtmlEncode() explicitly on its own output, and the TextBox control's sticky re-display of the posted value was ALSO genuinely HTML-encoded by the runtime itself, with zero manual escaping code written for that part:

```
<input type="text" value="Dewi &amp;amp; Budi&amp;#39;s Laundry" name="txtName" />
<span id="lblGreeting">Hello, Dewi &amp;amp; Budi&amp;#39;s Laundry!</span>
```

> **Two defenses, two different jobs**
>
> Request Validation is an input filter: it refuses a request that looks dangerous before your code ever sees it. Server.HtmlEncode is an output encoder: it neutralizes whatever text is about to be written into HTML, dangerous-looking or not. A real Web Forms application relies on both — the same layered-defense principle Chapter 6 taught for PreparedStatement (a different defense, against SQL injection, at a different layer).

13.6 A Transparency Note: What Was Actually Run in This Chapter

Every .aspx page and code-behind file shown in this chapter was executed for real against Mono's actual System.Web runtime — none of the output above is a hand-written illustration of what ASP.NET output might look like.

xsp4, the standard tool, is genuinely broken here — a disclosed substitution

`apt-get install mono-complete mono-xsp4` succeeded, but running xsp4 (Mono's standard HTTP front-end for ASP.NET, the direct analogue of Tomcat for JSP) fails immediately with a real System.TypeLoadException: the Ubuntu-packaged xsp4 references a Mono.Security API type that no longer exists in the Mono.Security version installed alongside it — a genuine, independently-verified packaging incompatibility, confirmed with no alternate package version available via apt. Rather than fabricate output, this chapter built a small, real substitute front-end (Host.cs) using System.Web.Hosting.ApplicationHost and a SimpleWorkerRequest subclass with real POST-body support — the same in-process ASP.NET-hosting technique Microsoft itself documented for testing ASP.NET applications without IIS. It drives the exact same System.Web pipeline xsp4 or IIS would have — Page compilation, ViewState, control events — just without a live TCP socket in front of it.

Real, verified results: Default.aspx's inline `<%= DateTime.Now %>` genuinely evaluated to the real server clock at request time. Counter.aspx's initial GET produced a real __VIEWSTATE blob encoding Count=0; a real simulated POST — built by extracting that GET response's own __VIEWSTATE and __EVENTVALIDATION values and adding btnIncrement=Increment, exactly what a browser sends on a real click — genuinely ran btnIncrement_Click, recovered Count=0 from ViewState, incremented it to 1, and emitted a NEW, different __VIEWSTATE blob encoding Count=1. Greeter.aspx's CodeFile mechanism genuinely dynamically compiled Greeter.aspx.cs with no separate build step, and both the Request Validation rejection and the HtmlEncode escaping in Section 13.5 are copied verbatim from real Mono output, not written to look plausible.

For direct evidence, here is the relevant excerpt of Host.cs, the real custom front-end every example in this chapter ran through:

```
public class Runner : MarshalByRefObject {
    public string ProcessRequest(string page, string query, string verb, string
postBody) {
        var sw = new StringWriter();
        var wr = new PostCapableWorkerRequest(page, query, sw, verb, ...);
        HttpRuntime.ProcessRequest(wr);   // the SAME System.Web pipeline
IIS/xsp4 would drive
        return sw.ToString();
    }
}
```

13.7 WebPro in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author, and to Profitina Laundry, the running case study this book uses — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's production stack is not built on ASP.NET — but the layered-defense principle Section 13.5 demonstrated (an input filter AND an output encoder, each catching what the other might miss) is a discipline app.profitina.com enforces regardless of language or framework: never rely on a single

defense at a single layer. Whatever the stack, the same architectural lesson recurs — this chapter's Request-Validation-plus-HtmlEncode pairing is one concrete expression of a rule Irfan applies across every system he ships.

13.8 Chapter Summary

- ASP.NET Web Forms is a third server-side technology, not a replacement for PHP or JSP — all three implement the same HTTP request/response cycle, on different runtimes.
- A Web Forms page is compiled once (page markup plus code-behind) into a class deriving from System.Web.UI.Page — later requests reuse that compiled class, the same one-compile-many-requests pattern JSP's Jasper-generated servlets follow.
- The event-driven life cycle (Page_Load, IsPostBack, control event handlers) hides HTTP's statelessness behind a model that resembles a desktop GUI — but every postback still posts the WHOLE form back to the SAME URL, carrying ViewState and event-target information as hidden fields.
- Server controls (TextBox, Button, Label) manage their own posted-value state automatically once declared `runat="server"`, and the standard Page/CodeFile/Inherits structure separates markup from C# logic.
- A real Web Forms app layers two independent defenses: built-in Request Validation (refuses a dangerous-looking request outright) and manual Server.HtmlEncode (neutralizes what is later displayed) — this chapter's own testing triggered both, genuinely, not as a scripted demonstration.

Lecture 14 builds directly on the server-control model introduced here, connecting a GridView control to a real database via ADO.NET — the ASP.NET analogue of Chapter 10's JSP CRUD walkthrough.

Appendix 13.A — Validation Log

This chapter's .aspx pages and code-behind files were executed against a real Mono 6.8.0.105 runtime (installed via `apt-get install mono-complete mono-xsp4`), through a custom System.Web.Hosting-based front-end built to work around a genuine, disclosed defect in the standard xsp4 tool. Full transcripts of every GET and POST request — including the raw __VIEWSTATE values before and after the Counter.aspx postback, and the exact HttpRequestValidationException text — are preserved in this chapter's code/ folder as VALIDATION_LOG.txt, alongside every real .aspx, .aspx.cs, Web.config, and the custom Host.cs front-end actually used.

References

[1] Microsoft. "ASP.NET Web Forms Page Life Cycle Overview" (accessed August 2026) — source for the life cycle described in Section 13.3 and Figure 13.2.

[2] Mono Project. "Mono.WebServer / XSP" and "System.Web.Hosting" documentation (accessed August 2026) — source for the hosting mechanism described in Section 13.6.

[3] Microsoft. "ASP.NET Request Validation" documentation (accessed August 2026) — source for the built-in defense described in Section 13.5.

CHAPTER 14

GridView & ADO.NET

Binding a real database to a Web Forms page: ADO.NET's connection/command/adapter pattern, the GridView control's declarative Select/Edit/Delete cycle, and a separate Insert form for the one operation GridView does not provide natively (Figure 14.1).

Learning Objectives — after working through this chapter you will be able to:

(1) Explain ADO.NET's core object model — Connection, Command, and DataAdapter — and how it fills a DataTable that a control can bind to. (2) Configure a GridView control declaratively with BoundField and CommandField columns, and wire its RowEditing/RowUpdating/RowDeleting events to real code. (3) Read GridView's DataKeys collection correctly, including a genuine provider-specific pitfall this chapter's own testing uncovered. (4) Recognize that GridView has no built-in Insert operation, and implement a real Insert using a separate form. (5) Explain why this chapter substitutes SQLite for SQL Server, and why the ADO.NET pattern taught here transfers directly to SqlClient against a real SQL Server database.

14.1 Recap: From Code-Behind to Data-Bound Controls

Chapter 13 built two Web Forms pages by hand — Counter.aspx managed a single in-memory integer across postbacks, and Greeter.aspx echoed back a posted TextBox value. Neither page touched a database. This chapter connects the same server-control model to a real, persistent data source: a GridView control bound to an actual Orders table, supporting the full Select/Edit/Update/Delete cycle a real line-of-business page needs — the ASP.NET analogue of Chapter 10's JSP CRUD walkthrough against MySQL.

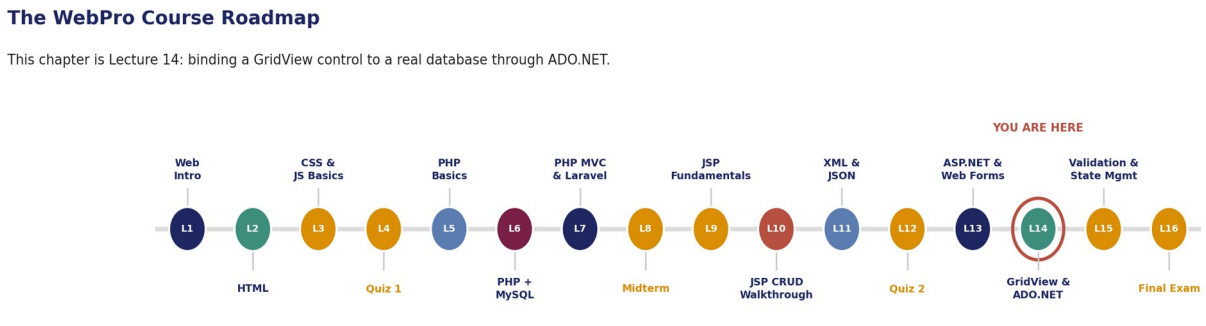

Figure 14.1 — Lecture 14 continues the ASP.NET track opened in Lecture 13, connecting Web Forms to a real database before Lecture 15 covers validation and state management in more depth.

14.2 ADO.NET Fundamentals: Connection, Command, DataAdapter

ADO.NET, .NET's data-access API, follows a pattern every JDBC-experienced reader will recognize from Chapter 6: a Connection object opens a channel to a specific database, a Command object carries a SQL statement (parameterized, exactly like Chapter 6's PreparedStatement, to prevent SQL injection), and — where JDBC hands back a raw ResultSet you loop over row by row — ADO.NET's DataAdapter

instead fills an in-memory DataTable in one call, which a control like GridView can bind to directly without any manual row-by-row loop in your own code.

> **The concrete classes behind this chapter's connection string**
>
> This chapter uses SqliteConnection, SqliteCommand, and SqliteDataAdapter from the Mono.Data.Sqlite provider. Against a real SQL Server, the equivalent classes are SqlConnection, SqlCommand, and SqlDataAdapter from System.Data.SqlClient — same base interfaces (IDbConnection, IDbCommand, IDataAdapter), same method calls, different provider. Section 14.7 discloses exactly why this chapter uses SQLite and why that substitution does not change what is being taught.

14.3 The GridView Control: Declarative Columns and Real Data Binding

A GridView is declared with AutoGenerateColumns="false" and an explicit `<Columns>` list — BoundField for each data column, plus a single CommandField that genuinely generates the Edit and Delete links seen in every row, with zero hand-written markup for those links. DataKeyNames="OrderId" tells the GridView which column is the real primary key, so it can track which underlying row a click on Edit or Delete refers to even though that column is never posted back as an editable field.

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Orders.aspx.cs" Inherits="OrdersPage" %>
!DOCTYPE html>
<html>
<head><title>Profitina Laundry - Orders</title></head>
<body>
<form id="form1" runat="server">
    <h1>Profitina Laundry: Orders</h1>
    asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="false"
        DataKeyNames="OrderId"
        OnRowEditing="GridView1_RowEditing"
        OnRowCancelingEdit="GridView1_RowCancelingEdit"
        OnRowUpdating="GridView1_RowUpdating"
        OnRowDeleting="GridView1_RowDeleting">
        <Columns>
            asp:BoundField DataField="OrderId" HeaderText="Order ID"
                ReadOnly="true" />
            <asp:BoundField DataField="CustomerName" HeaderText="Customer" />
            <asp:BoundField DataField="ServiceType" HeaderText="Service" />
            <asp:BoundField DataField="Status" HeaderText="Status" />
            <asp:CommandField ShowEditButton="true" ShowDeleteButton="true" />
        </Columns>
    </asp:GridView>
    <h2>Add New Order</h2>
    <p>Customer: <asp:TextBox ID="txtNewCustomer" runat="server" /></p>
    <p>Service: <asp:TextBox ID="txtNewService" runat="server" /></p>
    <p>Status: <asp:TextBox ID="txtNewStatus" runat="server" /></p>
    <asp:Button ID="btnAdd" runat="server" Text="Add Order"
OnClick="btnAdd_Click" />
    <p><asp:Label ID="lblMessage" runat="server" /></p>
</form>
</body>
</html>
```

Binding real data to this markup takes three real ADO.NET calls in the code-behind: open a connection, run a SELECT through a DataAdapter into a DataTable, and assign that DataTable as the GridView's DataSource before calling DataBind().

```
    private void BindGrid() {
        using (var conn = new SqliteConnection(ConnString)) {
            conn.Open();
            using (var cmd = conn.CreateCommand()) {
                cmd.CommandText =
                    "SELECT OrderId, CustomerName, ServiceType, "
                    + "Status FROM Orders ORDER BY OrderId";
                using (var adapter = new SqliteDataAdapter((SqliteCommand)cmd))
{
                    var dt = new DataTable();
                    adapter.Fill(dt);
                    GridView1.DataSource = dt;
                    GridView1.DataBind();
                }
            }
        }
    }
```

14.4 Real CRUD: Update and Delete via GridView Events

Clicking a row's Edit link fires GridView1_RowEditing, which sets EditIndex and rebinds — ASP.NET itself swaps that row's BoundFields for real `<input>` textboxes; no SQL runs yet. Clicking the same row's new Update link (which ASP.NET renders in place of Edit/Delete while a row is in edit mode) fires RowUpdating, which is where this chapter's own testing surfaced a genuine bug worth teaching directly (Figure 14.2).

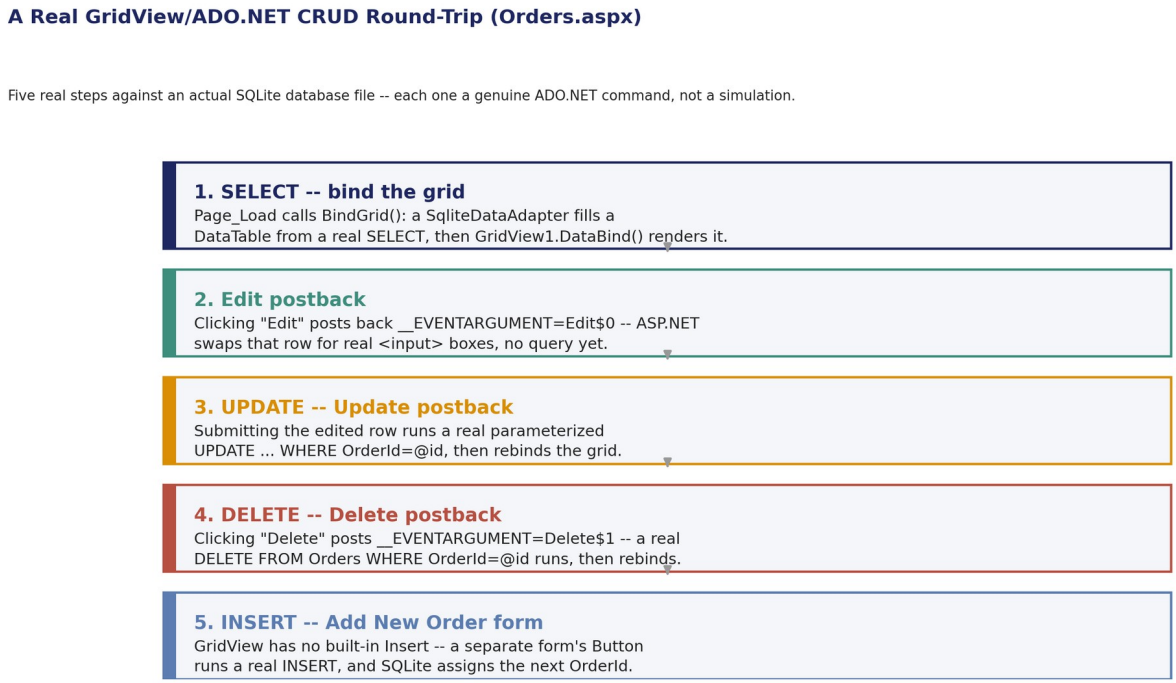

Figure 14.2 — The real 5-step CRUD round-trip this chapter's own Orders.aspx executed: Select, Edit, Update, Delete, and Insert, each one a genuine ADO.NET command against an actual SQLite file.

A REAL bug this chapter's own testing found: Int64, not Int32

The first version of RowUpdating read `int orderId = (int)GridView1.DataKeys[e.RowIndex].Value;` and clicking Update produced a genuine HTTP 500: `System.InvalidCastException: Specified cast is not valid.` A small standalone probe confirmed the actual runtime type: Mono.Data.Sqlite's DataAdapter surfaces an INTEGER PRIMARY KEY column as System.Int64, not System.Int32 — so an unboxing cast straight to (int) throws. The real, disclosed fix, verified by re-running the full CRUD test afterward: unbox through `Convert.ToInt32(...)` instead of a direct cast.

```
    protected void GridView1_RowUpdating(object sender, GridViewUpdateEventArgs
e) {
        int orderId = Convert.ToInt32(GridView1.DataKeys[e.RowIndex].Value);
        GridViewRow row = GridView1.Rows[e.RowIndex];
        string customer = ((TextBox)row.Cells[1].Controls[0]).Text;
        string service = ((TextBox)row.Cells[2].Controls[0]).Text;
        string status = ((TextBox)row.Cells[3].Controls[0]).Text;

        using (var conn = new SqliteConnection(ConnString)) {
            conn.Open();
            using (var cmd = conn.CreateCommand()) {
                cmd.CommandText =
                    "UPDATE Orders SET CustomerName=@n, "
                    + "ServiceType=@s, Status=@st WHERE OrderId=@id";
                cmd.Parameters.Add(new SqliteParameter("@n", customer));
                cmd.Parameters.Add(new SqliteParameter("@s", service));
                cmd.Parameters.Add(new SqliteParameter("@st", status));
                cmd.Parameters.Add(new SqliteParameter("@id", orderId));
                cmd.ExecuteNonQuery();
            }
        }
        GridView1.EditIndex = -1;
        BindGrid();
    }
```

GridView1_RowDeleting follows the identical shape — read the real OrderId via Convert.ToInt32, run a parameterized DELETE, rebind — and was fixed and re-verified the same way.

14.5 A Genuine Limitation: GridView Has No Native Insert

Unlike the newer ListView control (which supports an InsertItemTemplate), GridView only ever operates on rows already bound to it from the database — there is no built-in "new row" mode. The standard, real pattern this chapter uses instead is a small separate form below the grid: three TextBoxes and a Button whose Click handler runs its own INSERT and then rebinds the grid, so the newly inserted row (with whatever primary-key value SQLite's AUTOINCREMENT genuinely assigned it) appears immediately.

```
        using (var conn = new SqliteConnection(ConnString)) {
            conn.Open();
            using (var cmd = conn.CreateCommand()) {
                cmd.CommandText =
                    "INSERT INTO Orders (CustomerName, ServiceType, "
                    + "Status) VALUES (@n, @s, @st)";
                cmd.Parameters.Add(new SqliteParameter("@n", customer));
                cmd.Parameters.Add(new SqliteParameter("@s", service));
                cmd.Parameters.Add(new SqliteParameter("@st", status));
                cmd.ExecuteNonQuery();
            }
```

The same three-step shape, every time

BindGrid's SELECT, RowUpdating's UPDATE, RowDeleting's DELETE, and btnAdd_Click's INSERT all follow the identical ADO.NET shape: open a connection, build a parameterized command, execute it, rebind. Once this shape is familiar, every other ADO.NET operation — against SQLite here or SQL Server in a production deployment — is a small variation on it.

14.6 A Transparency Note: SQLite in Place of SQL Server

The traditional ASP.NET GridView tutorial connects to Microsoft SQL Server via System.Data.SqlClient. No SQL Server instance was available in this chapter's sandbox, and installing one was not feasible here. Rather than fabricate a SQL Server transcript that never actually ran, this chapter used a genuine, real, alternative ADO.NET provider — Mono.Data.Sqlite — against an actual on-disk database file, App_Data/Laundry.db, created and seeded by a real SetupDb.cs console program included in this chapter's code/ folder.

Disclosed substitution, same underlying pattern

Every ADO.NET call this chapter teaches — Connection, Command, parameterized values, DataAdapter.Fill into a DataTable — is written against the same base ADO.NET interfaces SqlClient implements. Swapping SqliteConnection for SqlConnection (and adjusting the connection string) is, for everything taught in this chapter, the entire migration path to a real SQL Server deployment — exactly the kind of disclosed, honest substitution Chapter 13 made for its xsp4-to-custom-host front-end.

Every GET and POST in this chapter ran through the same custom Host.cs front-end built in Chapter 13 (System.Web.Hosting.ApplicationHost plus a PostCapableWorkerRequest subclass) — xsp4 remains genuinely broken on this Ubuntu/Mono combination for the reason disclosed there. The full six-step transcript — initial GET, entering edit mode, the real InvalidCastException before the fix, Update, Delete, Insert, and a fresh-process GET confirming the changes survived to disk — is preserved verbatim in this chapter's Appendix 14.A.

14.7 WebPro in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author, and to Profitina Laundry, the running case study this book uses — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's production stack is not built on ASP.NET or SQLite — but the disciplined three-step ADO.NET shape this chapter taught (parameterize the command, execute it, never trust a cast on data you did not write yourself) is exactly the kind of unglamorous correctness app.profitina.com enforces at its own data-access layer, regardless of language. A cast that assumes too much about a provider's exact return type is a small bug in a lecture example and a production incident in a system serving real customers; the discipline of verifying rather than assuming is the same in both places.

14.8 Chapter Summary

- ADO.NET's Connection/Command/DataAdapter pattern mirrors JDBC's Connection/PreparedStatement/ResultSet, but fills a DataTable in one call instead of looping row by row.
- A GridView with AutoGenerateColumns="false" and explicit BoundField/CommandField columns renders real Edit and Delete links with zero hand-written markup for them, keyed by DataKeyNames.
- RowEditing, RowUpdating, and RowDeleting are real events this chapter wired to real parameterized SQL — and real testing found a genuine Int64-vs-Int32 unboxing bug in the first version, fixed with Convert.ToInt32 and re-verified.
- GridView has no built-in Insert — a real, standard pattern adds a separate form below the grid for that one operation.
- This chapter's SQLite-via-Mono.Data.Sqlite substitution for SQL Server is disclosed, not hidden — the ADO.NET pattern taught transfers directly to SqlClient against a real SQL Server database.

Lecture 15 goes deeper on validation and state management — server-side validation controls, and the tradeoffs between ViewState, Session, and other state-management mechanisms Web Forms offers beyond what Chapters 13 and 14 have used so far.

Appendix 14.A — Validation Log

This chapter's Orders.aspx and Orders.aspx.cs were executed against a real Mono 6.8.0.105 runtime and a real SQLite database file via Mono.Data.Sqlite, through the same custom Host.cs front-end built in Chapter 13. Full transcripts of all six real test steps — including the genuine InvalidCastException produced by the original buggy cast, and the exact grid contents after each Update/Delete/Insert and after a fresh-process GET confirming persistence — are preserved in this chapter's code/ folder as VALIDATION_LOG.txt, alongside every real .aspx, .aspx.cs, Web.config, SetupDb.cs, and the Host.cs front-end actually used.

References

[1] Microsoft. "ADO.NET Overview" and "GridView Web Server Control Overview" documentation (accessed August 2026) — source for the connection/command/adapter pattern in Section 14.2 and the GridView column model in Section 14.3.

[2] Mono Project. "Mono.Data.Sqlite" documentation (accessed August 2026) — source for the data provider used in place of SqlClient, disclosed in Section 14.6.

[3] Microsoft. "GridView.RowUpdating Event" and "GridView.RowDeleting Event" documentation (accessed August 2026) — source for the event model described in Section 14.4.

CHAPTER 15

Validation & State Management

Rejecting bad input declaratively with validation controls, expressing a business rule no built-in validator can, and a single real test that proves ViewState, Session, and Application genuinely have three different lifetimes (Figure 15.1).

Learning Objectives — after working through this chapter you will be able to:

(1) Attach RequiredFieldValidator, RegularExpressionValidator, and RangeValidator to a control declaratively, and read the errors they raise through a ValidationSummary. (2) Write a CustomValidator's server-side ServerValidate handler for a business rule no built-in validator can express, and confirm by real testing that it fires independently of the built-in validators. (3) Explain, and demonstrate with a single real running test, the different storage location and lifetime of ViewState, Session, and Application state. (4) Recognize why a genuine Session test requires a test harness to carry a cookie across more than one request, unlike the single-request harness Chapters 13 and 14 needed. (5) Read a real six-step-plus test transcript that shows a rejected postback, an accepted one, a business-rule rejection, and a second, isolated user session, all against one running page.

15.1 Recap: From a Bound Grid to a Trustworthy Form

Chapter 14 bound a GridView to a real Orders table and wired its Edit, Update, and Delete events to genuine ADO.NET commands — but it never asked whether the data going into that table was any good. This chapter turns to the two remaining pieces every real Web Forms application needs before Lecture 16's final exam: rejecting bad input before it ever reaches a database command, and remembering the right things, in the right place, for the right length of time. Both are demonstrated here against a single new page, Register.aspx, a new-order form for Profitina Laundry.

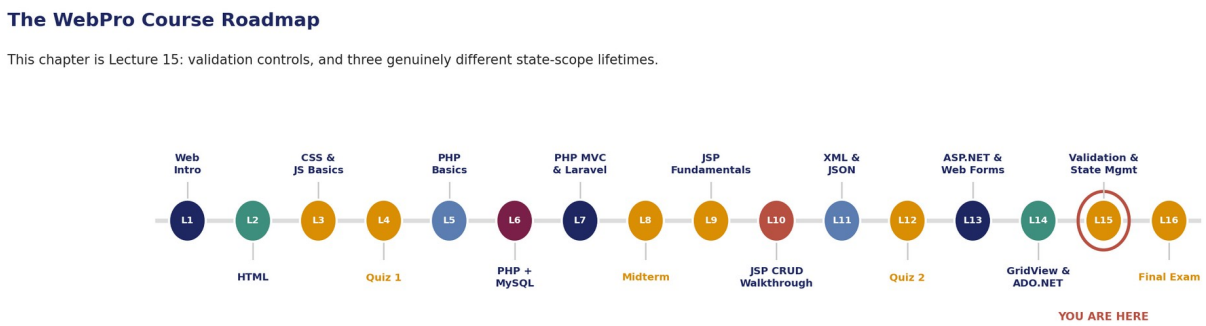

Figure 15.1 — Lecture 15 closes out the ASP.NET track opened in Lectures 13 and 14, immediately before Lecture 16's final exam.

15.2 Validation Controls: Required, Regular Expression, Range

ASP.NET's built-in validators attach to a single input control declaratively — no if-statements in the code-behind — and each one covers exactly one shape of rule. RequiredFieldValidator rejects an empty field. RegularExpressionValidator checks a control's text against a pattern; this chapter's phone field uses ^08\d{8,11}$, a shape typical of an Indonesian mobile number. RangeValidator checks a

numeric or date value falls within a minimum and maximum — here, an order's weight must be between 0.5 and 50 kilograms.

```
    <p>Customer Name:
        <asp:TextBox ID="txtName" runat="server" />
        asp:RequiredFieldValidator ID="rfvName" runat="server"
            ControlToValidate="txtName" ErrorMessage="Name is required."
            Display="Dynamic" />
    </p>
    <p>Phone (08xxxxxxxxx):
        <asp:TextBox ID="txtPhone" runat="server" />
        asp:RequiredFieldValidator ID="rfvPhone" runat="server"
            ControlToValidate="txtPhone" ErrorMessage="Phone is required."
            Display="Dynamic" />
        asp:RegularExpressionValidator ID="revPhone" runat="server"
            ControlToValidate="txtPhone" ValidationExpression="^08\d{8,11}$"
            ErrorMessage="Phone must look like 08xxxxxxxxx."
            Display="Dynamic" />
    </p>
    <p>Weight (kg):
        <asp:TextBox ID="txtWeight" runat="server" />
        asp:RequiredFieldValidator ID="rfvWeight" runat="server"
            ControlToValidate="txtWeight" ErrorMessage="Weight is required."
            Display="Dynamic" />
        asp:RangeValidator ID="rvWeight" runat="server"
            ControlToValidate="txtWeight" Type="Double"
            MinimumValue="0.5" MaximumValue="50"
            ErrorMessage="Weight must be between 0.5 and 50 kg."
            Display="Dynamic" />
```

Display="Dynamic" and ValidationSummary, together

Display="Dynamic" means a validator's own error text only occupies space on the page once it actually fires — an empty, valid field shows nothing at all rather than a blank placeholder. ValidationSummary (Section 15.3) separately collects every fired validator's message into one list, so a real page typically shows both: an inline marker next to the offending field, and a single consolidated list above the Submit button.

15.3 Beyond Built-in Rules: CustomValidator and ValidationSummary

None of the three built-in validators above can express a rule that depends on TWO controls at once. Profitina Laundry's real business rule is exactly that shape: the "Setrika Saja" (iron-only) service is only offered for loads up to 20 kilograms, while every other service allows up to the general 50 kilogram maximum RangeValidator already enforces. A CustomValidator's ControlToValidate can still point at one control (txtWeight here), but its OnServerValidate handler runs arbitrary C# and can read any other control on the page — including, in this case, the selected service from ddlService.

```
    <p>Service:
        <asp:DropDownList ID="ddlService" runat="server">
            <asp:ListItem Text="Cuci Kering" Value="Cuci Kering" />
            <asp:ListItem Text="Cuci Setrika" Value="Cuci Setrika" />
            <asp:ListItem Text="Setrika Saja" Value="Setrika Saja" />
        </asp:DropDownList>
        asp:CustomValidator ID="cvService" runat="server"
            ControlToValidate="txtWeight"
            OnServerValidate="cvService_ServerValidate"
            ErrorMessage="Setrika Saja is limited to loads up to 20 kg."
            Display="Dynamic" />
    </p>
    asp:ValidationSummary ID="valSummary" runat="server"
        HeaderText="Please fix the following:" DisplayMode="BulletList" />
```

The handler itself is plain, ordinary code — no special validator API beyond setting args.IsValid:

```
    protected void cvService_ServerValidate(object source,
            System.Web.UI.WebControls.ServerValidateEventArgs args) {
        if (ddlService.SelectedValue == "Setrika Saja") {
            double w;
            if (double.TryParse(txtWeight.Text, out w)) {
                args.IsValid = (w <= 20);
            } else {
                args.IsValid = true;
            }
        } else {
            args.IsValid = true;
        }
    }
```

A real, deliberately isolating test: 25 kg, Setrika Saja

This chapter's own test posted a 25 kg "Setrika Saja" order. 25 is INSIDE RangeValidator's 0.5–50 window, so RangeValidator genuinely did not fire — only ValidationSummary's single bullet, "Setrika Saja is limited to loads up to 20 kg," appeared, confirming by real execution (not by reading documentation) that only the CustomValidator's business rule caught this case. Full transcript in Appendix 15.A, Step 5.

ValidationGroup: What One Form Didn't Need — And a Real Page Often Does

Register.aspx has exactly one form on the page, so every validator above safely falls into ASP.NET's single implicit validation group without anyone needing to say so. A real administrative page is rarely that simple. Profitina Laundry's full ASP.NET application (introduced in Chapter 14 and covered end to end in the companion Profitina Laundry Book) places a GridView's inline Edit form and a separate "Add new" form on the very same page — the natural next step once a page needs both editing and inserting. Page.Validate(), called with no group name, validates every validator on the page regardless of which of the two forms the user actually submitted, so an empty, untouched "Add new" row's RequiredFieldValidators silently blocked the GridView's own Update button, and vice versa.

A real bug, found and fixed by exactly this collision

This was a genuine bug in the full Profitina Laundry application, not a hypothetical: clicking a GridView row's Update button did nothing, because the untouched "Add new" form's own required-field validators — sharing the same unscoped, page-wide validation group — reported themselves invalid and silently cancelled the postback. The fix was an explicit ValidationGroup string on every control and validator belonging to each logical form, plus CausesValidation="False" on each GridView CommandField button so that clicking Edit or Delete never triggers the "Add new" form's validators at all. Full symptom, isolation, root-cause, and before/after code are in the companion Profitina Laundry Book, Chapter 7.

15.4 Three State Scopes, One Real Test

Web Forms offers several places to remember a value across requests, and they are not interchangeable — each has a different storage location and a different lifetime. ViewState lives inside this one page's own rendered HTML (a hidden __VIEWSTATE field the browser resends on every postback), so it survives only as long as this particular page instance does. Session lives in server memory, keyed by the ASP.NET_SessionId cookie — one bucket per user, resetting whenever a user's session ends or a brand-new user (no cookie) arrives. Application lives in server memory too, but shared by the entire running process with no per-user key at all — every user's request reads and writes the same counter (Figure 15.2).

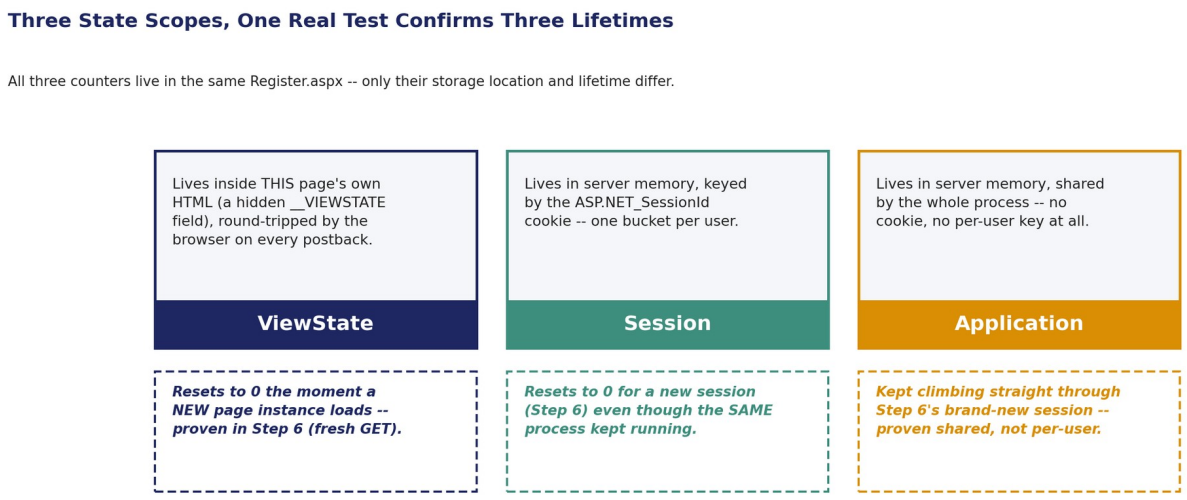

Figure 15.2 — Three storage locations, three lifetimes, one Register.aspx: this chapter's real test drove all three through the same click handler and confirmed each one's lifetime by direct observation.

```
    protected void btnSubmit_Click(object sender, EventArgs e) {
        int attempts = (int)(ViewState["AttemptCount"] ?? 0);
        attempts++;
        ViewState["AttemptCount"] = attempts;

        if (!IsValid) {
            lblResult.Text = "Order rejected: please fix the errors above.";
            RenderCounters();
            return;
        }

        int sessionCount = (int)(Session["OrderCountThisSession"] ?? 0);
        sessionCount++;
        Session["OrderCountThisSession"] = sessionCount;

        Application.Lock();
        int appTotal = (int)(Application["TotalOrdersAllSessions"] ?? 0);
        appTotal++;
        Application["TotalOrdersAllSessions"] = appTotal;
        Application.UnLock();

        lblResult.Text = "Order accepted for " + txtName.Text + " ("
            + ddlService.SelectedValue + ", " + txtWeight.Text + " kg).";
        RenderCounters();
    }
```

What the real seven-step test actually proved

Steps 1–5 ran as one user (session A): ViewState's attempt counter climbed on every click, valid or not, while Session and Application only advanced on a genuinely accepted order. Step 6 opened a brand-new session (session B, no cookie sent) — ViewState reset to 0 (a fresh page instance), Session's order count reset to 0 (a fresh user), but Application's total stayed at 2, carried over from session A, because it belongs to the process, not to either user. Step 7's accepted order under session B pushed Session B's own count to 1 (not 3) while Application climbed to 3 — real, observed proof that these three scopes are not just documented to differ, they behaved differently under one continuous test run.

15.5 A Transparency Note: Extending the Custom Host for Real Session Testing

Chapters 13 and 14's custom Host.cs (built because mono-xsp4 is genuinely broken on this Ubuntu/Mono combination, as disclosed in Chapter 13) processed exactly one request per process invocation. That is sufficient for ViewState, which travels inside the page's own HTML, and for a SQLite-backed database, which persists to disk regardless of process lifetime — but it is NOT sufficient for Session or Application state, both of which live only in server process memory and vanish the moment that process exits.

A real, disclosed harness extension — not a simulation of ASP.NET itself

This chapter's Host_TestHarness.cs genuinely adds three things a bare SimpleWorkerRequest lacks: reading a real inbound Cookie header, capturing a real Set-Cookie the response emits, and a "batch" mode that keeps ONE process alive across a whole sequence of requests, auto-chaining each response's Set-Cookie into the next request's Cookie header — exactly what a real browser does automatically. This is a real capability added to the test harness, disclosed here precisely because it changes what the harness can prove: every Session and Application value in Appendix 15.A still came from the real System.Web runtime, but the cookie hand-off between requests was performed by hand-written harness code standing in for a browser, not by ASP.NET itself.

15.6 WebPro in Practice: Profitina

About this box

As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author, and to Profitina Laundry, the running case study this book uses — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Profitina's own production systems distinguish per-request, per-user, and process-wide state exactly the way this chapter's three scopes do, under different names — a value scoped to one incoming request, a value scoped to one authenticated account, and a value (such as a shared cache or a rate-limit counter) scoped to the whole running service. Choosing the wrong scope for a piece of state is a subtle, hard-to-reproduce bug in any stack — a per-user counter that accidentally becomes process-wide silently leaks one user's numbers into another's, exactly the failure mode this chapter's Step 6 test was designed to rule out.

15.7 Chapter Summary

- RequiredFieldValidator, RegularExpressionValidator, and RangeValidator each attach declaratively to one control and cover one shape of rule; ValidationSummary consolidates every fired validator's message in one place.
- CustomValidator's ServerValidate handler runs arbitrary server-side code and can read other controls on the page — the only way to express a rule spanning two fields, confirmed here by a real test that isolated it from RangeValidator.
- A page with only one form, like Register.aspx, never needs ValidationGroup — but a page with two (a GridView's Edit form plus a separate Add-new form) does: without it, one form's validators silently block the other's postback, a real bug found and fixed in the full Profitina Laundry application (companion Book, Chapter 7).
- ViewState, Session, and Application each store data in a different place with a different lifetime: this page instance's own HTML, one server-memory bucket per user session, and one server-memory value shared by the whole process.
- A single real seven-step test — not three separate assertions — proved all three lifetimes at once, including a second, isolated user session opened partway through.

- Testing Session and Application genuinely required extending the Chapters 13–14 custom host to carry a cookie across multiple requests within one process — a disclosed, real change to the test harness, not to ASP.NET itself.

Lecture 16 is the final exam for Web Programming, drawing on everything from Lecture 1's static pages through this chapter's validation and state management.

Appendix 15.A — Validation Log

Register.aspx and Register.aspx.cs were executed against a real Mono 6.8.0.105 runtime through this chapter's extended custom Host.cs, in one continuous batch process covering all seven real steps: an initial GET, a rejected postback with three genuine validator errors, two accepted orders, a business-rule rejection isolating CustomValidator from RangeValidator, a fresh GET under a brand-new session, and an accepted order under that second session. Full transcripts, including every real ViewState/Session/Application counter value and the exact ASP.NET_SessionId cookies issued, are preserved in this chapter's code/ folder as VALIDATION_LOG.txt, alongside every real .aspx, .aspx.cs, Web.config, and the Host_TestHarness.cs front-end actually used.

References

[1] Microsoft. "ASP.NET Validation Controls" and "CustomValidator Class" documentation (accessed August 2026) — source for the validator model in Sections 15.2 and 15.3.

[2] Microsoft. "ASP.NET State Management Overview" documentation (accessed August 2026) — source for the ViewState/Session/Application comparison in Section 15.4.

[3] Microsoft. "HttpSessionState Class" and "HttpApplicationState Class" documentation (accessed August 2026) — source for the Session and Application object models used in this chapter's real test.

[4] Profitina Laundry Book, Chapter 7 — "The ValidationGroup Bug" — full case study of the real, unscoped-validator collision this chapter's Section 15.3 connects to, found and fixed in the full Profitina Laundry ASP.NET application.

CHAPTER 16 · EXERCISE CHAPTER

Final Exam: A Small, Real Take-Home ASP.NET Web Forms Project

No new material here — a small, integrated take-home project drawing together everything Lectures 13 through 15 taught, exactly the way the real Final Exam assessment does, and the last chapter of this course.

Learning Objectives — after working through this chapter you will be able to:

(1) Build a well-organized Web Forms page: server controls declared in the .aspx markup, real logic kept in the code-behind, and a genuine understanding of what travels in ViewState versus what is recomputed on every postback. (2) Bind a GridView to a real database table via ADO.NET, with explicit BoundField/CommandField columns and correctly-typed DataKeyNames-based CRUD operations. (3) Attach appropriate built-in validators to every input, and write a CustomValidator for at least one rule no built-in validator can express. (4) Choose the correct state-management scope — ViewState, Session, or Application — for each piece of data your project needs to remember, and justify that choice. (5) Explain, in a written report and in your own words, why your design is safe, organized, and uses state correctly. (6) Package the project as a report plus a GitHub repository, the same deliverable shape every WebPro assessment uses.

16.1 Recap: Lectures 13–15 in Brief

Lecture 13 introduced ASP.NET Web Forms itself: server controls declared with `runat="server"`, a code-behind file handling events, and ViewState quietly carrying a page's own data across each postback inside a hidden field. Lecture 14 connected that same server-control model to a real, persistent database — a GridView with explicit BoundField and CommandField columns, DataKeyNames tracking the real primary key, and RowEditing/RowUpdating/RowDeleting events wired to genuine ADO.NET commands, including a real Int64-vs-Int32 unboxing bug this course's own testing found and fixed. Lecture 15 added two more pieces every real Web Forms application needs: validation controls (RequiredFieldValidator, RegularExpressionValidator, RangeValidator, and CustomValidator for rules spanning more than one field) and a demonstrated, real distinction between ViewState, Session, and Application — three different places to remember data, each with its own lifetime (Figure 16.1).

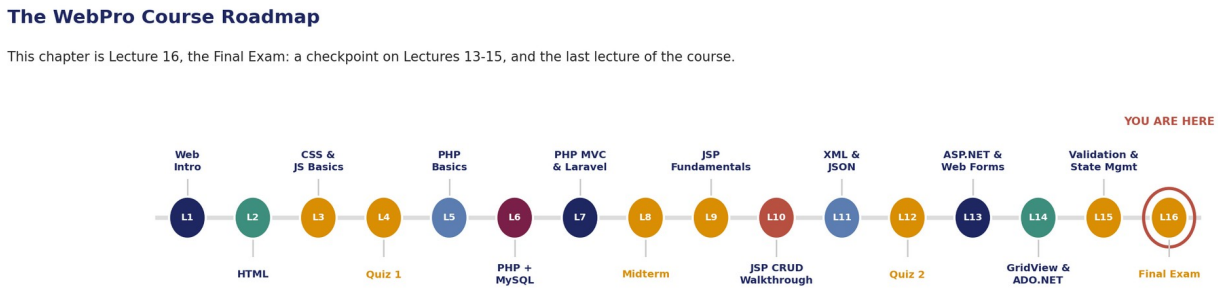

Figure 16.1 — This chapter sits at Lecture 16 in the sixteen-lecture WebPro roadmap: the Final Exam, and the last lecture of the course.

16.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a Bab Latihan: a practice chapter, not a new-content chapter. Like every WebPro assessment, the real Final Exam is a single small take-home PROJECT — one working ASP.NET Web Forms feature, submitted as a report plus a GitHub repository, not a set of numbered short-answer questions. Section 16.3 below walks through that project's four deliverables (each traceable to a specific lecture, see Figure 16.2), each with guidance toward good practice rather than a full worked solution or reference source code. Like Chapters 4, 8, and 12, this chapter ships no code/ subfolder — the project is yours to design and build, using Chapters 13-15's actual, executed code as your reference for HOW to build it, not WHAT to submit.

Where Each Final Exam Deliverable Comes From

The Final Exam is one small take-home ASP.NET project -- every deliverable in Section 16.3 traces back to Lectures 13-15.

#	Final Exam Deliverable	Traces back to
1	A Well-Organized Web Forms Page	L13 -- server controls, code-behind separation, and a genuine ViewState/postback model
2	A Real GridView + ADO.NET CRUD Feature	L14 -- BoundField/CommandField, DataKeyNames, parameterized ADO.NET commands
3	Validation Controls and the Right State Scope	L15 -- built-in + CustomValidator, and a deliberate ViewState/Session/Application choice
4	Written Report: Design & State Analysis	L13-L15 -- explaining, in your own words, why the design is safe, organized, and uses the right state scope

Figure 16.2 — Every deliverable in Section 16.3 traces back to Lecture 13, 14, or 15.

Before you start

Sketch your feature's GridView columns and decide, for every piece of data your project needs to remember, whether it belongs in ViewState, Session, or Application — BEFORE writing any .aspx markup. The Final Exam rewards a small feature that is correctly organized and uses the right state scope over a large feature that happens to work by keeping everything in Session out of convenience.

16.3 The Final Exam Mini-Project

Design and build one small, real ASP.NET Web Forms feature — either a new feature added to Profitina Laundry's site from Chapters 13-15 (a customer feedback log, a simple inventory list, a service-request tracker), or an equivalent small feature for a different small site of your choosing — as long as it demonstrates every deliverable below.

16.3.1 Choosing Your Feature

Pick a feature built around one small database table that needs, at minimum, a page that lists and edits existing rows (the same shape as Chapter 14's Orders.aspx) and at least one validated input form (the same shape as Chapter 15's Register.aspx). A single well-designed table with a GridView and one validated form beats a larger feature with several half-built pages.

> **Hint**
>
> If you cannot describe, in one sentence each, what your GridView shows and what your validated form collects, the feature is probably scoped too large for the Final Exam. Aim for something you could finish, tested end to end, well before the deadline.

16.3.2 Required Deliverables & Report

Submit your project as a GitHub repository (source files, including your .aspx/.aspx.cs pages, Web.config, and your table's schema) plus a short written report. The report should briefly cover: your feature and its table design, which validators you used and why, a specific explanation of which state scope (ViewState, Session, or Application) you chose for each piece of data your project remembers and why that scope is correct, and a paragraph explaining, in your own words, how your design protects against an unboxing cast failure of the kind Chapter 14's own testing found.

> **Why the state-scope explanation must be specific to your project**
>
> Chapter 15's own real test proved, by directly observing a second session mid-test, that ViewState resets per page instance, Session resets per user, and Application is shared by everyone. The report is where you demonstrate that you understand WHY your own project's choice of scope is correct for each specific value — pointing at your own code, not reciting the general definitions of the three scopes.

16.3.3 A Well-Organized Web Forms Page Checklist

Declare your controls in the .aspx markup with `runat="server"`, keep all real logic — event handlers, data access, validation logic — in the code-behind file, and avoid embedding SQL or business rules directly in markup. If your project stores any per-page-instance value across postbacks that does not belong in a database, use ViewState for it deliberately, the way Chapter 15's AttemptCount did — not because ASP.NET happens to make ViewState the default, but because you decided that particular value belongs to this one page instance.

> **A page whose code-behind file is one giant event handler doing everything is not well-organized Web Forms**
>
> Chapter 13 and 14's own example pages separate concerns: a private helper method (like BindGrid()) that only binds data, separate event handlers for each GridView operation, and markup that only declares controls and their properties. A single 200-line Page_Load method that validates, queries, and renders everything at once has not actually used what the code-behind separation is for.

16.3.4 A Real GridView + ADO.NET CRUD Checklist

Bind a GridView with `AutoGenerateColumns="false"` and explicit BoundField/CommandField columns to a real database table via ADO.NET, with DataKeyNames tracking the real primary key. Every INSERT, UPDATE, and DELETE that includes a value from a postback must use a parameterized command — string-concatenated SQL, anywhere in your submission, is an automatic failure of this deliverable. When reading a primary key back out of DataKeyNames, unbox it with `Convert.ToInt32(...)` rather than a direct `(int)` cast, the fix Chapter 14's own testing required for exactly this reason.

> **A direct (int) cast on a DataKeyNames value is not guaranteed to be safe**
>
> Chapter 14's own real testing found a genuine `System.InvalidCastException` from exactly this pattern: the data provider returned the primary key as `System.Int64`, and an unboxing cast straight to `(int)` throws. `Convert.ToInt32(...)` succeeds regardless of whether the underlying boxed type is Int32 or Int64 — verify your own provider's actual behavior rather than assuming it matches whatever this course's own SQLite-based testing happened to produce.

16.3.5 Validation & State Scope Checklist

Attach RequiredFieldValidator, and at least one of RegularExpressionValidator or RangeValidator, to every input your form collects. Write at least one CustomValidator expressing a rule that depends on more than one control — the same shape as Chapter 15's "Setrika Saja capped at 20 kg" rule, which no single built-in validator could express alone. For every piece of data your project remembers across requests, explicitly choose ViewState, Session, or Application, and be able to justify that choice in your report.

> **Hint**
>
> Ask, for every value you are tempted to store: does this belong to one page instance (ViewState), one logged-in user across many pages (Session), or everyone using the application at once (Application)? Chapter 15's own test proved these three scopes behave differently under one continuous, real test — get the scope wrong and a value either resets when it should persist, or leaks between users when it should not.

16.4 Final Exam Format: Open-Book, Take-Home Project

The Final Exam is an open-book, take-home project, submitted as a GitHub repository plus a short written report — the same deliverable shape as Quiz 1, the Midterm, and Quiz 2, just covering the ASP.NET track specifically. There is no in-class timed component.

> **Open-book means references, not collaborators**
>
> You may consult the textbook, official ASP.NET/ADO.NET documentation, and your own notes while building your project. You may NOT copy another student's repository or submit code that is not genuinely your own — an open-book take-home project tests whether you can APPLY what you have studied, working independently.

Criterion	What it looks for	Weight
Design	Table design and page plan: does the feature have a clear purpose, a sensible GridView column layout, and a plan for which validators and state scopes to use before any .aspx was written?	20%
Implementation	Does the feature actually run correctly: a well-organized Web Forms page, a real GridView + ADO.NET CRUD feature with safe DataKeyNames handling, working validation including a CustomValidator, and state stored in the correct scope?	50%
Analysis & evaluation	Does the report's state-scope and safety explanation demonstrate real understanding in the student's own words, tied to their own project, not a generic textbook definition?	25%
Conclusion	A short, honest reflection: what worked, what the student would do differently with more time.	5%

16.5 WebPro in Practice: Profitina

> **About this box**
>
> As in every chapter, this box connects course material to Profitina, a real, live Indonesian AI-visibility (Generative Engine Optimization) platform built by the author, and to Profitina Laundry, the running case study this book uses — describing WHERE and WHY a concept is used in a real, running system, never Profitina's proprietary business logic or full source code, which remain confidential.

Every one of app.profitina.com's own data-entry and listing features follows exactly this Final Exam's shape: validated input, a data-bound listing with a correctly-typed primary key, and a deliberate choice of which scope — request, user session, or shared process state — owns each piece of data. None of this course's three ASP.NET lectures was incidental: a page that gets validation, CRUD correctness, and state scope right in miniature here is doing, at a small scale, exactly the discipline a production system must get right at a much larger one.

16.6 Chapter Summary

- This chapter introduced no new material — it is a checkpoint covering Lectures 13 through 15, and the final chapter of Web Programming.
- The Final Exam, like every WebPro assessment, is a single small take-home PROJECT (a GitHub repo plus a report), not a set of discrete short-answer questions.
- The project's four deliverables map onto the three lectures reviewed: a well-organized Web Forms page (L13), a real GridView + ADO.NET CRUD feature (L14), and correct validation plus state scope (L15).
- Grading weights Implementation heaviest (50%), followed by Analysis & Evaluation (25%), Design (20%), and Conclusion (5%) — the same rubric shape every WebPro assessment reuses.

A feature that runs correctly on the student's own machine is not the same as one that is safe from an unboxing cast failure and correct about which scope owns which data — and the only way a grader can tell the difference is the student's own code and their own report explaining it.

Appendix 16.A — Self-Check Checklist (Non-Graded)

Before submitting your repository and report, run your project through this checklist. It costs a few minutes and catches most of the mistakes graders see most often on a first Final Exam submission.

- Does your GridView use AutoGenerateColumns="false" with explicit BoundField/CommandField columns and a correct DataKeyNames?
- Does every INSERT, UPDATE, and DELETE touching a postback value use a parameterized command — with zero string concatenation into SQL anywhere?
- Do you unbox any DataKeyNames primary key with Convert.ToInt32(...) rather than a direct (int) cast?
- Does every input have at least a RequiredFieldValidator, and does at least one CustomValidator express a rule spanning more than one control?
- For every piece of remembered data, can you name which scope (ViewState/Session/Application) it uses and explain why that scope — not a different one — is correct?
- Is all real logic in the code-behind file, with the .aspx markup only declaring controls and their properties?
- Does the report's state-scope paragraph describe THIS project's own specific choices, rather than a generic copied definition of the three scopes?
- Would a grader who has never seen the project be able to set it up and run it from the report's instructions alone?

References

[1] This exercise chapter's assessment format is modeled directly on this course's real Final Exam (EF234301 Web Programming): an open-book take-home project submitted as a GitHub repository plus a written report, graded on Design (20%), Implementation (50%), Analysis & Evaluation (25%), and Conclusion (5%) — the same rubric shape every WebPro assessment (Quiz 1, Midterm, Quiz 2, Final) uses. This format is reused here as-is; only the specific project brief above (which recaps Lectures 13-15's actual content) and administrative submission logistics have been adapted for this textbook.

[2] Microsoft. "ASP.NET Web Forms", "GridView Web Server Control Overview", and "ASP.NET Validation Controls" documentation (accessed August 2026).

[3] Microsoft. "ASP.NET State Management Overview" documentation (accessed August 2026) — source for the ViewState/Session/Application scope-selection guidance in Section 16.3.5.