Profitina Laundry

First Edition

Irfan Subakti

Profitina

profitina.com

TABLE OF CONTENTS

Preface 4

Preface — Why This Book Exists 6

1.1 One Case Study, Many Purposes 6
1.2 Grounded in a Real Business 6
1.3 Who This Book Is For 7
1.4 A Commitment to Honesty 7
1.5 How This Book Is Organized 8

The Real Story Behind Profitina Laundry 9

2.1 A Real Business Behind a Teaching Case 9
2.2 Why the Details Are Adjusted, Not Exact 9
2.3 A Short Chapter, On Purpose 9

How a Laundry Business Actually Works 11

3.1 The Physical Flow: From Intake to Pickup 11
3.2 The People: Roles in a Small Laundry 11
3.3 What Has to Be Tracked, and Why 12
3.4 The Rules a Real Laundry Enforces, Whether It Knows It or Not 12
3.5 From Operations to Software 13

The Data Model: Turning a Business Into a Schema 14

4.1 Where This Schema Comes From 14
4.2 Customer and Employee: Two Tables, One Shape 14
4.3 The Self-Referencing Supervisor 15
4.4 Service, and the Rule Against Deleting What's Already Been Billed 15
4.5 Bill: Where Customer, Employee, and Service Meet 15
4.6 A Note on Status: One Coarse Field, Three Timestamps 16
4.7 Two Design Choices Worth Naming Honestly 16
4.8 From Schema to Software 17

Four Technologies, One Business: An Architecture Tour 18

5.1 Static (Chapters 2-3): The Front End, on Its Own 18
5.2 PHP (Chapters 5-7): The First Real Backend 18
5.3 JSP (Chapters 9-11): Full CRUD, Jakarta EE Style 18
5.4 ASP.NET (Chapters 13-15): The Richest Implementation 18
5.5 Side by Side 19
5.6 Why Four Implementations, Not One 19
5.7 What's Next 19

Inside the ASP.NET Application 20

6.1 A Shared Master Page 20
6.2 Four CRUD Pages, One Shared Pattern 20
6.3 NewBill.aspx: Where the Business Rules Live 20
6.4 Three Places to Keep a Count: ViewState, Session, and Application 21
6.5 The Cross-Platform Reality, Stated Plainly 21
6.6 What's Next 22

The ValidationGroup Bug: A Debugging Case Study 23

7.1 The Symptom 23
7.2 The First, Wrong Instinct 23

7.3 Isolating the Problem..... 23
7.4 The Real Root Cause..... 24
7.5 Why This Bug Is Easy to Write and Hard to Notice..... 24
7.6 The Fix..... 24
7.7 Verifying the Fix Was Real, Not Assumed..... 25
7.8 The Broader Lesson..... 25
7.9 What's Next..... 26

Getting Started: Running Profitina Laundry Yourself..... 27
8.1 Static (Chapters 2-3)..... 27
8.2 PHP (Chapters 5-7)..... 27
8.3 JSP (Chapters 9-11)..... 27
8.4 ASP.NET (Chapters 13-15)..... 28
8.5 Where to Find All of This on Disk..... 29

Lessons Beyond the Code..... 30
9.1 For Educators: A Case Study Built to Be Adapted, Not Just Read..... 30
9.2 For Business Owners: What "Digitizing" Actually Involves..... 30
9.3 For Students: Reading Real Code, Including Its Flaws..... 31
9.4 One General Lesson, Stated Once More..... 31
9.5 What's Left..... 31

Closing: Where Profitina Laundry Goes Next..... 32
10.1 What Is Genuinely Finished..... 32
10.2 What Remains a Tracked Follow-Up, Named Honestly..... 32
10.3 Where Profitina Laundry Goes Next..... 32
10.4 A Closing Word..... 33

Preface

This book is a field report, not a brochure. It was written while the system it describes was being built, and it keeps the working notes rather than tidying them away.

Most textbook case studies are invented purely to demonstrate a syntax feature. Profitina Laundry took the other path: it is grounded in a real small business the author actually owned and ran, rebuilt honestly across four different technology stacks instead of once.

Profitina Laundry opens by explaining why one grounded case study, built four times, is worth more than four invented ones built once each, then tells the real (deliberately brief, deliberately adjusted for privacy) story behind the business. It explains how a laundry business actually operates independent of any software, then turns that operational reality into a data model — a schema shaped by real roles, real rules, and real reporting needs. An architecture tour walks through all four technologies the business was built on, with a full chapter going inside the ASP.NET application specifically, and a dedicated debugging case study around one real, instructive bug: the ValidationGroup bug. The book closes practically — how to run Profitina Laundry yourself — and reflectively, with lessons that outlast the code and where the project goes next.

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

- See how a real small business's operational reality — intake, processing, ready-for-pickup, payment — gets turned into a database schema.
- Compare the same business logic implemented across four different technology stacks, side by side.
- Go inside a working ASP.NET application built for a real, non-invented business problem.
- Follow a genuine debugging case study — the ValidationGroup bug — from symptom to root cause to fix.
- Run Profitina Laundry yourself as a hands-on reference implementation.
- Take away lessons about grounding a case study in reality that apply well beyond this one project.

Who this book is written for: students working through the Web Programming (WebPro) or Framework-Based Programming (FBP) courses, who meet this exact system as their running case study across many lectures; educators looking for a case study they can adapt to their own course; and small-business owners and the general public curious what "digitizing" a small service business actually requires in practice.

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

Preface — Why This Book Exists

A small laundry business, a real story behind it, and the case for writing all of this down properly.

> **What this book is**
>
> This is the story and the technical documentation of Profitina Laundry: a fictional laundry business, built as a real, working software system four separate times, once per server-side technology taught in the Web Programming (WebPro) course, and later a fifth time in the Framework-based Programming (FBP) course. It is written for three audiences at once — students learning to build real systems, educators looking for a grounded, honest case study, and business owners curious what "digitizing" a small service business actually involves in practice.

1.1 One Case Study, Many Purposes

Most textbook case studies are invented purely to illustrate a syntax feature: a "Book" and "Author" table to demonstrate a JOIN, a "Product" and "Order" table to demonstrate a foreign key. They are disposable, forgotten the moment the lesson ends. Profitina Laundry was built to be the opposite of that: one coherent business, modeled once with real care, and then implemented completely and honestly across every technology a course needs to teach — so that what a student learns about the business's data model, its rules, and its rough edges on page one is still true on page two hundred.

That decision has a cost. Building one running application is work. Building the same one four times, in four different technologies, each one real enough to expose real bugs and real design tradeoffs, is a much larger undertaking. This book exists because that work is worth writing down properly, rather than letting it live only as scattered source files across a project folder — the same instinct that leads a real engineering team to write a design document, not just ship the code.

1.2 Grounded in a Real Business

Profitina Laundry is not a purely invented scenario. It is built on the author's own experience owning and running a laundry business, together with a business partner and a small staff of employees, in an earlier chapter of life. The system in this book — the customers, the employees, the services, the bills, the day-to-day rhythm of intake, wash, fold, and pickup — is inspired directly by that real experience, adjusted and simplified here and there for teaching purposes: names, numbers, and specific details are fictionalized, but the shape of the business, the roles people actually held, and the operational rhythm they actually followed are real.

> **Why this matters more than it might seem to**
>
> A case study invented purely for a syntax lesson tends to model the database, not the business — it has just enough structure to demonstrate a JOIN or a foreign key, and no more. A case study grounded in a real business tends to surface real complexity in the right places: a supervisor who is themselves an employee (a genuine self-referencing relationship, not a contrived one), a service that cannot simply be deleted once a customer has been billed for it (a genuine referential-integrity concern, not an artificial rule), a front-counter employee who should be assigned new walk-in customers by actual current workload, not always the same name at the top of a list. Every one of these appears in this book's schema and code not because it makes a good textbook example, but because it is what a real laundry business's records actually look like.

1.3 Who This Book Is For

Three overlapping audiences will find something useful here, and this book tries not to write only for the first of them.

- Students and self-learners working through the WebPro or FBP courses, who will meet this exact system as their running case study across many lectures, and who can use this book as the single place where the whole system — not just one chapter's slice of it — is explained coherently.
- Educators looking for a case study they can adapt to their own course, who may be interested less in Profitina Laundry's specific code and more in the design decisions behind it: why one canonical schema, why four independent technology implementations instead of one, why real execution testing and honest bug disclosure rather than idealized clean-room examples.
- Small business owners and the general public curious what "digitizing" a small service business like a laundry actually requires in practice — not as a sales pitch for a specific product, but as an honest, worked example: what data a business like this actually needs to track, what rules that data needs to enforce, and what a real, running booking-and-billing system for a business like this actually looks like end to end.

1.4 A Commitment to Honesty

Every technical claim in this book is backed by something that was actually run, not merely described. Where a page says a piece of code works, that claim comes from a real execution log: a real database, a real web server, a real HTTP request and response, inspected directly rather than assumed. Where this project's own development process turned up a real bug — and it did, more than once — this book says so plainly, including exactly how the bug was found and exactly how it was fixed, because that is at least as valuable a lesson as a line of code that happened to work on the first try.

> **What this book will not do**
>
> It will not claim a technology "just works" cross-platform when it genuinely does not — classic ASP.NET Web Forms, covered in Chapters 6 and 8, is a clear example: it is a real, still-widely-deployed technology, but it was never ported to modern cross-platform .NET, and this book says exactly that rather than quietly hoping the reader does not notice on a Mac. It will not present a clean, bug-free development story where the real one had a genuine bug in it. Both of those shortcuts are common in technical writing, and both would make this book less useful to exactly the readers it is written for.

1.5 How This Book Is Organized

Chapter 2 tells the (deliberately brief) real story behind Profitina Laundry. Chapter 3 explains how a laundry business actually operates, independent of any software, for readers who have never run one. Chapter 4 turns that operational reality into a data model — the actual database schema this project settled on, and why. Chapter 5 tours all four technology implementations at a high level, and Chapter 6 goes deep on the richest of the four, the ASP.NET Web Forms application, including the exact business rules it enforces. Chapter 7 is a single, focused case study of a real bug this project found in its own code, how it was diagnosed, and how it was fixed — a chapter about debugging methodology as much as about ASP.NET specifically. Chapter 8 is a practical getting-started guide for readers who want to run any of the four implementations themselves, on Windows, macOS, or Linux. Chapter 9 steps back and draws out what this project teaches beyond its own code, for educators and for business readers. Chapter 10 closes with where Profitina Laundry goes next.

Readers only interested in the technology can start at Chapter 4 and skip Chapters 2-3 without losing anything essential to the code. Readers mainly interested in the real-world business grounding, or in what a project like this can teach a small business owner, may find Chapters 2, 3, and 9 the most rewarding, and can treat the code-heavy chapters in between as optional depth rather than required reading.

CHAPTER 2

The Real Story Behind Profitina Laundry

A brief, honest note on where this business came from.

> **In short**
>
> Profitina Laundry was created based on the author's own experience owning and managing a laundry business together with a business partner and a group of employees, in an earlier chapter of life. Naturally, adjustments were made here and there for the purposes of this book and its courses. That is the whole of it — and, deliberately, this chapter does not go much further than that.

2.1 A Real Business Behind a Teaching Case

It would have been easy to invent Profitina Laundry from nothing — to sketch a plausible-sounding small business, give it a name, and build a schema around whatever seemed pedagogically convenient. That is, in fact, how most textbook case studies are made, and Chapter 1 already explained why that shortcut tends to produce case studies that model only what the lesson needs and nothing more.

Profitina Laundry took the other path. Its owner-and-partner structure, its front-counter and back-of-house employees, its services, its customers, and the day-to-day rhythm of taking in laundry and returning it clean are not invented from a blank page — they are drawn from a real laundry business the author actually owned and ran, together with a partner and a small staff, at an earlier point in life. The system this book documents is that business, translated into software, with the specific details adjusted for teaching.

2.2 Why the Details Are Adjusted, Not Exact

Two things are true at once, and this book wants to be honest about both. First, the business Profitina Laundry is grounded in was real: real customers, real employees, real day-to-day decisions about pricing, staffing, and workload. Second, this book does not reproduce that business's exact names, exact numbers, or exact history — the names in this book's sample data, the specific pricing figures, and the precise sequence of events are fictionalized and adjusted, sometimes for the ordinary privacy reasons any real business's story deserves, and sometimes simply because a teaching example benefits from round numbers and clean illustrations more than a real ledger ever does.

What survives the adjustment is the shape of the thing: the roles people genuinely held, the rules the business genuinely needed enforced, and the operational rhythm it genuinely followed. That is also, not coincidentally, exactly what a data model needs to get right — which is why Chapter 4's schema is able to lean on this real grounding for decisions like a self-referencing supervisor relationship or a service that resists deletion once a customer has been billed for it, rather than inventing those rules for effect.

2.3 A Short Chapter, On Purpose

Readers expecting a longer origin story — specific dates, a business name, a city, a blow-by-blow account of how the partnership came together or what happened along the way — will not find one here, and that is a deliberate choice rather than an omission. The lesson this book has to teach does not depend on those specifics, and offering invented specifics in their place would blur the line between what is genuinely drawn from experience and what is fabricated for narrative color. This chapter says only what needs to be said: the business is real, the software is new, and the adjustments in between are ordinary ones.

The next chapter picks up from here in a more useful direction for readers building the software: it explains, independent of any particular technology, how a laundry business like this one actually operates day to day — the operational reality that Chapter 4 then turns into a concrete data model.

CHAPTER 3

How a Laundry Business Actually Works

The operational reality behind the software, before any code is written.

Before looking at a single line of code or a single database table, it helps to understand the business those things exist to serve. A laundry business is simple enough to describe in a few paragraphs, yet rich enough to need real software once it grows past a certain size — which is exactly the range this book is interested in.

3.1 The Physical Flow: From Intake to Pickup

A load of laundry moves through a small number of stages between the moment a customer drops it off and the moment they collect it, and nearly every laundry business, however it is organized, follows some version of the same sequence.

- Intake — a customer brings in a bag or basket of items, or has it picked up. An employee weighs it (laundries typically price by the kilogram) or counts pieces, notes which service applies (a simple wash-and-fold, a dry-clean, an iron-only order, and so on), and records who the order belongs to.
- Processing — the laundry is washed, dried, and folded or pressed according to the service ordered. In a small business this is done by hand or with a handful of machines, by whichever employees are on shift; in a larger one it may move through dedicated stations.
- Ready for pickup — once processing is complete, the order waits for the customer, and the business needs some way to know which orders are actually ready versus still in progress, especially once more than a handful of orders are in the shop at once.
- Pickup and payment — the customer returns, the order is handed over, and payment is collected (or, for a regular customer with a running account, added to a tab settled periodically).

Chapter 4's schema names these stages directly: a Bill's status moves through exactly this sequence, and that sequence is not an arbitrary software design choice — it is simply what happens at the front counter, written down.

3.2 The People: Roles in a Small Laundry

A laundry business small enough to run without dedicated software still usually has more than one kind of role in it, and the relationships between those roles matter more than they might first appear to.

- An owner or a small partnership of owners, who set prices, decide which services to offer, and are ultimately responsible for the business.
- A small staff of employees, who staff the front counter, handle intake, and do the washing, drying, folding, and pressing itself.
- Often, one employee acts as a shift supervisor or senior staff member for the others — someone who is still an employee themselves, but who other employees report to day to day.

> **Why the supervisor relationship matters to the data model**
>
> That last point — a supervisor who is also an employee — is easy to overlook in an invented case study, because inventing a business from scratch rarely produces an org chart with real texture to it. A real laundry business, however, almost always has this shape: a senior employee supervises others without being a separate kind of person in the system. That is precisely why Chapter 4's Employee table has a self-referencing supervisor relationship — one employee record pointing to another employee record — rather than a separate, simpler "manager" concept. The real business shaped the schema, not the other way around.

3.3 What Has to Be Tracked, and Why

Even a very small laundry business, run entirely on paper, ends up tracking roughly the same handful of things, because each one answers a question the business gets asked constantly.

What is tracked	The question it answers
Customers	Whose laundry is this, and how do we reach them when it's ready?
Employees	Who took this order, who is doing the work, and who is responsible for it?
Services	What was ordered, and what should it cost?
Bills / orders	What is the status of this specific order right now, and what does the customer owe?

Every one of these four categories shows up, under one name or another, in every one of Profitina Laundry's four software implementations — the flattened `orders` table the JSP track teaches in Chapter 10 folds Services and Bills together into a single row, while the ASP.NET track's `Customer` / `Employee` / `Service` / `Bill` tables (Chapter 6) keep all four separate and related. Both are valid answers to the same underlying operational reality; Chapter 5 compares them directly.

3.4 The Rules a Real Laundry Enforces, Whether It Knows It or Not

A business run on paper enforces its own rules through habit and common sense, even when nobody has written them down as formal policy. Digitizing the business means making those unwritten rules explicit — and a few of Profitina Laundry's rules are worth calling out here because they come directly from how a real laundry actually has to operate, not from software design taste.

- A service that has already been billed to a customer cannot simply be deleted from the price list, because the bill still refers to it — deleting it would leave that bill pointing at nothing. A real laundry would never let a price-list change quietly erase what a customer was already charged for, and neither should the software.
- New walk-in customers are best assigned to whichever front-counter employee currently has the lightest load, not always to the same senior employee by default — a real shop balances work across whoever is on shift, and a booking system that ignores this quickly produces one overworked employee and several idle ones.

- An order's status can only move forward through the intake-to-pickup sequence in one direction under normal operation; an order does not go back to "received" after it has been marked ready, because that is not something that happens to real laundry.

These are not abstract software requirements invented to make a good textbook example. They are the ordinary common sense of running a laundry counter, and Chapters 4 and 6 show exactly how each one is enforced in the data model and in the ASP.NET application's code.

3.5 From Operations to Software

With this operational picture in view, the next chapter turns it into something a database can hold: a concrete schema, with concrete tables, columns, and relationships, built to say exactly what Sections 3.3 and 3.4 above described — no more, and no less.

CHAPTER 4

The Data Model: Turning a Business Into a Schema

Customer, Employee, Service, and Bill — and the real rules behind each relationship.

Chapter 3 described four categories a laundry business has to track — customers, employees, services, and the bills that tie them together — and a handful of unwritten rules a real laundry enforces without ever calling them "business rules." This chapter turns that operational picture into the actual schema this project settled on: the canonical `Bill` / `Customer` / `Employee` / `Service` model that the ASP.NET track (Chapters 6-8) implements directly, and that the JSP track's flattened `orders` table (Chapter 10) is a deliberate simplification of.

4.1 Where This Schema Comes From

This is not a schema invented for this book. It is adapted, table for table and column for column, from the project's original entity-relationship design — a Vertabelo diagram drawn against the real business Chapter 2 described, later exported as MySQL DDL (`All SQLs.txt`). The version shown in this chapter is the SQLite dialect the ASP.NET application actually runs against; the JSP track's MariaDB database uses the same logical shapes for `Customer`, `Employee`, `Service`, and `Bill` with only dialect-level differences (`ENUM` instead of `TEXT` with a `CHECK` constraint, for instance).

4.2 Customer and Employee: Two Tables, One Shape

A customer and an employee are, at the level of what a laundry needs to know about a person, almost the same kind of record: a name, a gender, a place of origin, a birthdate, an address, a phone number, and an email address. The schema reflects that directly.

```
CREATE TABLE Customer (
    id        TEXT PRIMARY KEY CHECK (length(id) <= 5),
    name      TEXT NOT NULL,
    gender    TEXT NOT NULL DEFAULT 'f' CHECK (gender IN ('f','m')),
    place     TEXT NOT NULL,
    birthdate TEXT NOT NULL,
    address   TEXT NOT NULL,
    phone     TEXT NOT NULL,
    enrolled  TEXT NOT NULL,
    email     TEXT NOT NULL,
    password  TEXT NOT NULL CHECK (length(password) <= 5),
    credit    NUMERIC(10,2) NOT NULL DEFAULT 0
);
```

`Employee` carries the same personal columns, plus two that mark it as specifically an employee record: `section` (which part of the business this employee works in — `transaction`, the front counter, or `wash`, the processing side) and `started` (when their employment began, mirroring `Customer.enrolled`). Both tables also carry a `password` column capped at five characters, and both IDs (`c0001`, `e0001`, and so on) are capped at five characters as well — both are inherited unchanged

from the original 2020 Vertabelo design, and Section 4.7 addresses the first of those two honestly rather than glossing over it.

4.3 The Self-Referencing Supervisor

Chapter 3 named a real operational fact: in a small laundry, one employee often supervises the others while remaining an employee themselves, rather than occupying some separate "manager" category. The schema expresses that exactly as it is in reality — as an `Employee` table with a foreign key back onto itself.

```
    spv_id    TEXT NULL,
    FOREIGN KEY (spv_id) REFERENCES Employee(id)
        ON DELETE SET NULL ON UPDATE CASCADE
```

Why ON DELETE SET NULL here, specifically

If a supervising employee ever leaves the business and their record is removed, the employees who reported to them should not be deleted along with them — they simply become unsupervised until a new supervisor is assigned. `ON DELETE SET NULL` encodes exactly that: deleting `e0001` (Borat Sagdiyev, who supervises `e0002` and `e0003` in the seed data below) clears their `spv_id` back to `NULL` rather than cascading the deletion or blocking it outright. A real laundry business does not fire an entire team because its supervisor moved on, and the schema agrees.

4.4 Service, and the Rule Against Deleting What's Already Been Billed

`Service` is the simplest table in the schema — an id, a description, and a price — but it is where Chapter 3's delete-protection rule becomes visible as an actual constraint rather than a policy someone has to remember to enforce by hand.

```
CREATE TABLE Service (
    id          TEXT PRIMARY KEY CHECK (length(id) <= 5),
    description TEXT NOT NULL,
    price       NUMERIC(7,2) NOT NULL
);
```

`Bill.Service_id` references `Service.id` with `ON DELETE RESTRICT` — the database itself refuses to delete a service that any existing bill still refers to. Chapter 6 shows what this looks like from the ASP.NET application's side: an attempt to delete an in-use service is caught and reported back to the user as a clear error, rather than allowed to fail as an unhandled database exception, or worse, silently allowed to orphan a bill.

4.5 Bill: Where Customer, Employee, and Service Meet

`Bill` is the table everything else exists to support — one row per order, tying together who dropped it off, who is handling it, and what was ordered.

```
CREATE TABLE Bill (
    id          TEXT PRIMARY KEY CHECK (length(id) <= 5),
    arrived     TEXT NOT NULL,
    finished    TEXT NULL,
    departed    TEXT NULL,
    status      TEXT NOT NULL DEFAULT 'queued'
                CHECK (status IN ('queued','finished','departed')),
    rack        TEXT NOT NULL,
    unit        INTEGER NOT NULL,
    payment     NUMERIC(10,2) NOT NULL,
    Customer_id TEXT NOT NULL,
    Employee_id TEXT NOT NULL,
    Service_id  TEXT NOT NULL,
    FOREIGN KEY (Customer_id) REFERENCES Customer(id) ON DELETE RESTRICT,
    FOREIGN KEY (Employee_id) REFERENCES Employee(id) ON DELETE RESTRICT,
    FOREIGN KEY (Service_id)  REFERENCES Service(id)  ON DELETE RESTRICT
);
```

All three of `Bill`'s foreign keys use `RESTRICT`, for the same underlying reason as Section 4.4: a customer, employee, or service that any existing bill still references cannot simply vanish out from under that bill. `rack` records where the physical laundry is physically stored while it waits (a real, physical detail a real laundry has to track, seldom mentioned in invented case studies), and `unit` records how many pieces or kilogram-units the order covers, feeding directly into `payment`.

4.6 A Note on Status: One Coarse Field, Three Timestamps

`Bill.status` moves through only three values — `queued`, `finished`, `departed` — which is coarser than the four-stage intake-to-pickup flow Chapter 3 described in prose (intake, processing, ready, picked up). That is not an inconsistency to paper over; it is a genuine, disclosed difference in how the two tracks model the same operational reality. The ASP.NET schema folds "intake" and "processing" into a single `queued` state, and instead of a fourth status value for "ready," it records the moment an order became ready directly, as the `finished` timestamp, alongside `arrived` and `departed`. The JSP track's `orders` table (Chapter 10) takes the opposite approach: four explicit status values (`received`, `washing`, `ready`, `picked_up`) and no separate timestamp columns at all.

Both are legitimate ways to model the same four-stage reality; Chapter 5 compares the two tracks' schemas side by side, and the project's own README and both schema files disclose this difference explicitly rather than treating one as simply more correct than the other.

4.7 Two Design Choices Worth Naming Honestly

- Five-character IDs (`c0001`, `e0001`, `C4W`, `b0001`) are a direct inheritance from the original 2020 Vertabelo design, not a limit this book would choose today for a growing business — a real laundry chain outgrowing five-character IDs would need to widen this column, and that is a normal, expected kind of schema migration, not a design flaw specific to Profitina Laundry.
- Both `Customer.password` and `Employee.password` store a plaintext value capped at five characters. This is carried over from the original design for schema fidelity, and neither column is ever read from or written to by any page in the application — the course this schema supports never taught a login feature, so these two columns exist for completeness

only. A real system storing actual credentials must hash them (with bcrypt, PBKDF2, or Argon2) and must never cap a password's length this way; this book says so plainly rather than let a reader copy this pattern into a production system.

4.8 From Schema to Software

With the schema now concrete, the next chapter steps back out to the full picture: how this same underlying business logic and data model is expressed four different times, in four different technologies, across the Static, PHP, JSP, and ASP.NET tracks — what each one shares, and what each one, honestly, does differently.

CHAPTER 5

Four Technologies, One Business: An Architecture Tour

What Static, PHP, JSP, and ASP.NET each teach, and why they aren't built alike.

Profitina Laundry is not one application — it is the same business, expressed independently four separate times, once per server-side technology the Web Programming (WebPro) course teaches. This chapter tours all four at a high level, as independent, self-contained projects rather than four front-ends to one shared backend, before Chapter 6 goes deep on the richest of the four.

5.1 Static (Chapters 2-3): The Front End, on Its Own

The earliest implementation is deliberately backend-free: plain HTML5, CSS3, and vanilla JavaScript, with no server-side code and no database at all. Its purpose is narrow and specific — teaching semantic HTML and client-side scripting in isolation, before a server enters the picture — and it succeeds at that precisely because it stays this simple. It reads its handful of services from a static `services.json` file via `fetch()`, which is also the one place it needs any kind of HTTP server at all, since browsers block `fetch()` on a bare `file://` page for security reasons.

5.2 PHP (Chapters 5-7): The First Real Backend

The PHP track is where Profitina Laundry first touches a real database. A visitor's contact-form submission is validated, then written through PDO prepared statements into a `messages` table in MariaDB/MySQL, and a separate `view-messages.php` page reads it back. It is intentionally narrower in scope than the JSP and ASP.NET tracks — one form, one table — because Chapters 5-7 are where the course introduces server-side form handling and database access for the first time, not where it teaches full CRUD.

5.3 JSP (Chapters 9-11): Full CRUD, Jakarta EE Style

The JSP/Servlet track is Profitina Laundry's first full create-read-update-delete implementation: `orders-list.jsp`, `order-create.jsp`, `order-edit.jsp`, and `order-delete.jsp`, backed by a single flattened `orders` table (Chapter 4, Section 4.6 compares its status model to the ASP.NET track's). Chapter 11 adds the same data as machine-readable XML and JSON exports, `orders-xml.jsp` and `orders-json.jsp`, with a client-side `fetch()` demo consuming the JSON directly. It runs on Apache Tomcat 10, which matters specifically because Tomcat 10 is the first version to use the modern `jakarta.servlet.*` namespace rather than the older `javax.servlet.*` one — code written for Tomcat 10 will not run unmodified on Tomcat 9 or earlier, and this book says so plainly rather than leaving readers to discover it the hard way.

5.4 ASP.NET (Chapters 13-15): The Richest Implementation

The ASP.NET Web Forms track is Profitina Laundry's most complete implementation, and the only one of the four to target the full canonical `Bill` / `Customer` / `Employee` / `Service` schema from Chapter

4 directly, with dedicated CRUD pages for each of the four entities plus a purpose-built booking page, `NewBill.aspx`, that enforces the load-balanced employee assignment rule Chapter 3 described. It is also the one implementation among the four with a genuine, disclosed cross-platform limitation, which Chapter 6 addresses head-on rather than glossing over.

5.5 Side by Side

Track	Chapters	Technology	Data model
Static	2-3	HTML5 / CSS3 / JavaScript	None (static services.json)
PHP	5-7	PHP + PDO + MySQL/MariaDB	Single messages table
JSP	9-11	JSP/Servlets + Tomcat 10 + MariaDB	Flattened orders table
ASP.NET	13-15	ASP.NET Web Forms + SQLite	Bill / Customer / Employee / Service

5.6 Why Four Implementations, Not One

A course teaching four different server-side technologies could, in principle, teach all four against a single shared case study built once and simply re-skinned each time. Profitina Laundry deliberately does not do this, for a reason worth stating directly: each technology is taught at a different point in the course, at a different level of sophistication, and building each implementation to genuinely fit its chapter block — rather than forcing every track onto one schema from day one — lets each track teach what its chapters are actually about. The Static track can stay a front-end-only exercise without an awkward, unused backend bolted on; the PHP track can stay focused on one form and one table without pretending to be full CRUD before the course is ready to teach that.

> **What this design choice costs, honestly**
>
> The cost is exactly the schema divergence Chapter 4, Section 4.6 already disclosed: the JSP track's orders table and the ASP.NET track's Bill/Customer/Employee/Service schema are not the same shape, and a student moving from Chapter 10 to Chapter 13 will notice that the data model changed under them. That is a real, tracked follow-up for this project (harmonizing the two, alongside updating the affected chapters' book text, not just their code) rather than an oversight — see this book's closing chapter for where that stands.

5.7 What's Next

Chapter 6 goes deep on the ASP.NET track specifically: its pages, its business rules, and the three-tier ViewState/Session/Application state model its booking page relies on. Chapter 7 then tells the story of a real bug this project found in its own ASP.NET code, and exactly how it was diagnosed and fixed.

CHAPTER 6

Inside the ASP.NET Application

Pages, business rules, and three ways to keep a count.

Of Profitina Laundry's four implementations, the ASP.NET Web Forms application is the most complete: it targets the full canonical schema from Chapter 4, and it is the only track with a purpose-built booking workflow rather than plain CRUD alone. This chapter walks through what it actually does, page by page, and the business rules it enforces along the way.

6.1 A Shared Master Page

All six pages — Default (the dashboard), Services, Customers, Employees, Bills, and NewBill — share a single `Site.master` layout: consistent navigation, consistent styling (`app.css`), and a single place to change the site's look without touching every page individually. This is Web Forms' own answer to the kind of shared-layout problem every one of Profitina Laundry's other three tracks solves differently (a shared `<header>` in the Static and PHP tracks' plain HTML, `index.jsp`'s own chapter-by-chapter navigation in the JSP track).

6.2 Four CRUD Pages, One Shared Pattern

Services.aspx, Customers.aspx, Employees.aspx, and Bills.aspx each follow the same layout: a GridView listing existing rows with inline Edit and Delete commands, above a separate "Add a New X" form for creating new ones. Each enforces the rule Chapter 4 built into the schema itself — attempting to delete a Service, Customer, or Employee that an existing Bill still references is caught and reported back to the user as a clear message, rather than allowed to surface as a raw database exception.

Employees.aspx additionally exposes the self-referencing supervisor relationship from Chapter 4, Section 4.3 directly: its Add-and-Edit forms include a supervisor dropdown populated from the Employee table itself, and deleting a supervising employee is reflected immediately in every employee record that pointed to them, exactly as the schema's `ON DELETE SET NULL` rule specifies.

6.3 NewBill.aspx: Where the Business Rules Live

NewBill.aspx is Profitina Laundry's booking page, and it is where Chapter 3's operational rules stop being abstract and become enforced application logic.

- Server-computed payment — the amount a customer owes is calculated on the server from the selected service's price and the number of units entered, never trusted from client-submitted input. A customer cannot influence what they are charged by tampering with the form.
- Load-balanced employee assignment — when a new bill is created, the front-counter ("transaction") employee with the fewest currently-queued bills is assigned automatically, rather than defaulting to the same employee every time. This is Chapter 3's staffing observation, enforced in code rather than left to whichever employee happens to be logged in.

- Returning-customer recognition — a customer is looked up by phone number before a new Customer row is created; a phone number already on file reuses the existing record instead of creating a duplicate. A real laundry counter recognizes a regular customer by more than their exact name spelling, and this page does the same.
- A service-specific quantity limit, enforced by a CustomValidator — orders for the Iron Only ("SS", Setrika Saja) service are capped at a sensible maximum number of units; an order for that service submitted well above the cap (validated at 25 units) is rejected before it reaches the database, while a normal-sized order for the same service (validated at 15 units) is accepted. A single iron-only order for dozens of units is very unlikely to be genuine, and rejecting it early catches a data-entry mistake before it becomes a bill.

6.4 Three Places to Keep a Count: ViewState, Session, and Application

NewBill.aspx also does something most CRUD pages never need to: it keeps three separate running counters of how many bills have been booked, each stored in a different one of ASP.NET Web Forms' state-management mechanisms, specifically so the difference between the three is visible rather than theoretical.

Mechanism	Counts	Survives
ViewState	Bookings made on this specific page instance	Postbacks to the same page, resets on a fresh page load
Session	Bookings made by this specific visitor	Every request from the same browser session, resets when the session ends
Application	Bookings made by everyone, across every visitor	The lifetime of the running application itself

This is a genuinely useful thing to see in a real page rather than a slide: a single successful booking increments all three counters at once, but they diverge the moment a second visitor books something — that second booking increments Session and Application for that visitor's own session, while the first visitor's Session counter stays exactly where it was, and only Application moves for both of them. Verifying this actually happens (not merely assuming ASP.NET's documentation is right) took real cross-request testing infrastructure, described briefly in Chapter 7's testing notes.

6.5 The Cross-Platform Reality, Stated Plainly

> **This is a real limitation, not an oversight**
>
> Classic ASP.NET Web Forms — the `System.Web` API this application is written against — was never ported to modern, cross-platform .NET (.NET 5/6/7/8 and beyond). It runs natively only on Windows with IIS or IIS Express and the .NET Framework. On Linux or macOS, the only way to run it at all is Mono, an independent, third-party re-implementation of the .NET Framework, and Mono's own lightweight web host is known to be unreliable for Web Forms specifically on some Mono/OS combinations. This book states that plainly rather than claiming a false cross-platform story, and Chapter 8's setup guide gives Windows 11 as the first-class path, with Mono and a Windows virtual machine as the two honest options for macOS and Linux.

Because of this, testing this application's real behavior on this project's own Linux development environment required a small, purpose-built substitution — never shipped as part of the actual application — described honestly in this track's own `VALIDATION_LOG.txt` and referenced again in Chapter 7.

6.6 What's Next

Chapter 7 is a single, focused story: a real bug this project's own ASP.NET code had, hiding in plain sight in every one of the four CRUD pages described in Section 6.2, and the exact process that found it, explained it, and fixed it.

CHAPTER 7

The ValidationGroup Bug: A Debugging Case Study

A silent GridView failure, and the process that actually found it.

Every real-execution claim in this book rests on something that actually happened, and this chapter is about the most instructive thing that happened while building the ASP.NET track: a genuine bug, found in this project's own code, that silently broke every GridView's Update button on every one of the four CRUD pages Chapter 6 described. This chapter tells that story in full — not because the bug itself is exotic, but because how it was found is a debugging lesson worth having in a book, not just in a commit message.

7.1 The Symptom

Clicking "Update" on a row in the Services, Customers, Employees, or Bills GridView appeared to do nothing. No error appeared on screen. The page posted back — a network round-trip clearly happened — and the grid simply re-rendered with the row's original, unedited values still in place. Nothing about the visible behavior pointed at a specific cause: no exception, no validation message, no red text anywhere.

7.2 The First, Wrong Instinct

The first thing worth checking looked, briefly, like the explanation: the rendered HTML for the Update button's postback call differed depending on a setting called `CausesValidation`. With it left at its default (`true`), the button posts back via its own uniquely generated control ID with an empty argument; changed to `false`, it posts back through the GridView's own ID with an indexed argument identifying which row and command fired. That is a real, legitimate difference in how ASP.NET renders the two configurations — but it is a rendering detail, not a bug, and chasing it as the root cause would have been a dead end. It is documented in this track's `VALIDATION_LOG.txt` specifically so it is not mistaken for the actual defect by a future reader making the same first guess.

7.3 Isolating the Problem

Rather than keep guessing inside the full application, the next step was to build the smallest possible thing that could reproduce the symptom: a minimal standalone page with nothing but a GridView and an Update command. That minimal page worked perfectly — `RowUpdating` fired every time, values persisted correctly. The same minimal page, rebuilt against a shared master page matching Site.master, still worked. Only when an unrelated, unfilled `RequiredFieldValidator` was added elsewhere on the same page — modeling the real app's separate "Add a New X" form sitting alongside the grid — did the symptom reproduce exactly: the Update command silently stopped firing.

> **Why building a minimal reproduction mattered here**
>
> The full application has master pages, multiple GridViews, multiple forms, and a fair amount of surrounding code — plenty of places a bug could plausibly hide, and plenty of noise to sift through while looking for it. A minimal, deliberately narrowed-down reproduction removes all of that noise at once: once the bare page with nothing but a GridView and one unrelated validator reproduced the exact symptom, the search space collapsed from "somewhere in four pages of code" to "this specific interaction between a GridView command button and a validator it has nothing to do with."

7.4 The Real Root Cause

Every CRUD page pairs a GridView (with an editable `CommandField`) with a separate "Add a New X" form on the same page, and — in the code as originally written — neither one specified a `ValidationGroup`. ASP.NET Web Forms' default behavior, when no group is specified anywhere on a page, is to validate every validator on the page together as one single implicit group, regardless of which form a button visually belongs to.

The GridView's Update button has `CausesValidation="true"` by default. Clicking it therefore triggered `Page.Validate()` against that single default group — which included the Add-form's own `RequiredFieldValidator` controls, naturally empty during a grid-only edit, since the user was editing an existing row, not filling out the Add form at all. Validation failed. And ASP.NET Web Forms' standard behavior when a `CausesValidation="true"` button's validation fails is to skip that button's Click/Command handler entirely — which meant `RowUpdating` never ran, with no exception and no visible error, because from the framework's point of view nothing had gone wrong: it had correctly declined to run a handler behind a failed validation check.

```
<%-- Before the fix: no ValidationGroup anywhere on the page --%>
<asp:GridView ID="GridView1" runat="server" >
  <Columns>
    <asp:CommandField ShowEditButton="true" ShowDeleteButton="true" />
  </Columns>
</asp:GridView>

<%-- The unrelated Add form, same page, same default validation group --%>
<asp:RequiredFieldValidator runat="server" ControlToValidate="txtNewName"  />
<asp:Button ID="btnAdd" runat="server" Text="Add" />
```

7.5 Why This Bug Is Easy to Write and Hard to Notice

Nothing about this code looks wrong on a read-through. Both the GridView and the Add form are correct, idiomatic Web Forms markup on their own; the defect only exists in the interaction between the two, and only because ASP.NET's page-wide default validation group is an implicit, easy-to-forget behavior rather than something a developer typically writes out loud. A page with just a GridView, or just an Add form, would never show this symptom — it takes both, on the same page, with no explicit grouping, which is exactly the shape every one of Profitina Laundry's four CRUD pages happened to have.

7.6 The Fix

The fix is to make the validation grouping explicit rather than relying on the implicit default: every validator, `ValidationSummary`, and button belonging to the Add form gets an explicit `ValidationGroup`, and the GridView's `CommandField` (along with any other control that can trigger an unwanted validation, such as Bills.aspx's auto-postback service dropdown) gets `CausesValidation="False"`, since grid row edits do not need page-level validation triggered on their behalf at all.

```
<%-- After the fix --%>
asp:CommandField ShowEditButton="true" ShowDeleteButton="true"
                 CausesValidation="False" />

asp:RequiredFieldValidator runat="server" ControlToValidate="txtNewName"
                           ValidationGroup="AddService" ... />
asp:Button ID="btnAdd" runat="server" Text="Add"
           ValidationGroup="AddService" />
```

Applied across Services.aspx, Customers.aspx, Employees.aspx, and Bills.aspx (each with its own group name — `AddService`, `AddCustomer`, `AddEmployee`, `AddBill`), this fix restores exactly the behavior the pages were always meant to have: editing a grid row no longer touches the unrelated Add form's validators at all, and `RowUpdating` fires every time.

7.7 Verifying the Fix Was Real, Not Assumed

A fix that merely looks right is not the same as a fix confirmed to work. Every affected page was re-tested end to end after the change: adding a row, editing a row through the grid, deleting a row, and reading the underlying SQLite database directly after each operation to confirm the change actually persisted, rather than trusting the page's own rendered response. All four pages passed, across every operation, and the one remaining detail Section 7.2 flagged — the Update button's postback pattern changing from an own-ID form to a GridView-indexed form once `CausesValidation` became `False` — was confirmed as an expected rendering consequence of the fix, not a new problem, and the project's own test scripts were updated to recognize both patterns rather than assume only one.

Confirming the fix did not break NewBill.aspx's ViewState/Session/Application counters (Chapter 6, Section 6.4) needed something extra: those three counters only diverge meaningfully across multiple requests in the same running application, which a simple one-request-per-call test cannot exercise. That required building a small addition to this project's test harness — a mode that keeps one ASP.NET application instance alive across a whole stream of requests, so Session and Application state could be observed changing for real, across real successive requests, rather than assumed from documentation.

7.8 The Broader Lesson

The specific defect here — an unscoped `ValidationGroup` colliding across two unrelated forms on the same page — is a genuine, well-known ASP.NET Web Forms pitfall, worth knowing on its own. But the more durable lesson is the process: a vague, unhelpful symptom ("nothing happens, no error") was resolved not by staring harder at the full application, but by building a smaller and smaller reproduction until the exact minimal trigger was visible, and the fix was then verified against a real

database and real HTTP behavior rather than trusted on inspection alone. That process generalizes far beyond ASP.NET, and readers debugging an entirely different technology stack will recognize the same shape of investigation.

7.9 What's Next

Chapter 8 is a practical guide to running any of Profitina Laundry's four implementations for yourself, on Windows 11, macOS, or Linux.

CHAPTER 8

Getting Started: Running Profitina Laundry Yourself

Windows 11, macOS, and Linux setup for all four implementations.

This chapter is a practical companion, not a reference manual — it gets each of Profitina Laundry's four implementations running on Windows 11, macOS, or Linux, with the same honest platform notes Chapters 5 and 6 already gave in passing. The full, always-current version of these instructions ships alongside the code itself, in the top-level `README.md` of the application bundle; this chapter condenses it for readers of the book.

8.1 Static (Chapters 2-3)

No backend, no build step, and no server required in the strict sense — opening `Static/index.html` directly in any modern browser works identically on all three platforms. The one exception is the page's `fetch()` call, which browsers block on a bare `file://` page for security reasons; serving the folder instead avoids that entirely.

```
cd Static
python3 -m http.server 8000
# then open http://localhost:8000/index.html
```

This one command is identical on Windows 11, macOS, and Linux, since Python 3 ships the same built-in server everywhere. VS Code's "Live Server" extension, `npx serve`, or PHP's own `php -S` all work the same way, if any of those happen to already be installed.

8.2 PHP (Chapters 5-7)

1. Install PHP 8.x and a MySQL-compatible server: XAMPP on Windows 11 (bundles PHP, MariaDB/MySQL, and Apache in one installer); `brew install php mariadb` on macOS, or MAMP as a bundled alternative; `sudo apt install php php-mysql mariadb-server` on Ubuntu/Debian.
2. Create the database and load the schema (identical on every platform once step 1 is done):

```
mysql -u root -p -e "CREATE DATABASE profitina_laundry CHARACTER SET utf8mb4;
  CREATE USER 'webpro'@'127.0.0.1' IDENTIFIED BY 'webpro_pass123';
  GRANT ALL PRIVILEGES ON profitina_laundry.* TO 'webpro'@'127.0.0.1'; FLUSH
PRIVILEGES;"
mysql -u webpro -pwebpro_pass123 -h 127.0.0.1 profitina_laundry < PHP/schema.sql
```

3. Serve the folder with PHP's own built-in server and open the contact form:

```
cd PHP
php -S localhost:8091
# then open http://localhost:8091/contact-form.html
```

8.3 JSP (Chapters 9-11)

4. Install a JDK (11 or later) and Apache Tomcat 10.x specifically — Tomcat 10 uses the `jakarta.servlet.*` namespace this code targets; Tomcat 9 and earlier use the older, incompatible `javax.servlet.*` namespace and will not run this code unmodified. On Windows 11: a JDK such as Eclipse Temurin, plus the Tomcat 10 ZIP distribution (no installer needed). On macOS: `brew install openjdk@21 tomcat@10`. On Ubuntu/Debian: `sudo apt install openjdk-21-jdk tomcat10`.
5. Put MariaDB Connector/J on Tomcat's shared classpath (its `lib/` folder on every platform), either downloaded directly from MariaDB's Connector/J releases, or via `sudo apt install libmariadb-java` on Ubuntu/Debian, which installs it straight into the system Tomcat's shared `lib/`.
6. Create the database and load the schema:

```
mysql -u root -p -e "CREATE DATABASE profitina_laundry CHARACTER SET utf8mb4;
  CREATE USER 'webpro'@'127.0.0.1' IDENTIFIED BY 'webpro_pass123';
  GRANT ALL PRIVILEGES ON profitina_laundry.* TO 'webpro'@'127.0.0.1'; FLUSH
PRIVILEGES;"
mysql -u webpro -pwebpro_pass123 -h 127.0.0.1 profitina_laundry < JSP/schema.sql
```

7. Copy the whole JSP/ folder into Tomcat's webapps/ directory as "profitina", start Tomcat, and open http://localhost:8080/profitina/index.jsp.

8.4 ASP.NET (Chapters 13-15)

> **Read this before choosing a path**
>
> Classic ASP.NET Web Forms was never ported to modern cross-platform .NET — it only ever runs natively on Windows with IIS or IIS Express and the .NET Framework, or under Mono (a third-party, independent re-implementation) on Linux or macOS. Overselling "cross-platform" here would be dishonest, so this book does not.

Windows 11 — the fully supported, first-class path

8. Install Visual Studio 2022 with the "ASP.NET and web development" workload — this installs IIS Express and the .NET Framework targeting pack together.
9. Install the System.Data.SQLite.Core NuGet package into the project.
10. Create the SQLite database once, from Schema.sqlite.sql, using any SQLite tool (the sqlite3 CLI, or DB Browser for SQLite).
11. Open the project in Visual Studio and press F5 — IIS Express serves it directly, with no separate web server install needed.

macOS — two honest options, neither of them "native"

- Mono (`brew install mono`): a genuine option, but Mono's lightweight web host is known to be unreliable for Web Forms on some Mono/OS combinations — this project's own testing hit exactly that, and built a small test-only harness rather than relying on it (see Chapter 7). Budget real troubleshooting time.
- A Windows virtual machine (Parallels, UTM, or VMware, on Apple Silicon or Intel): install Windows 11 inside the VM and follow the Windows steps above. This is the path most likely to work on the first try, and what most real-world Mac-using Web Forms developers have historically done.

Linux

Same Mono path as macOS above, since Mono is itself cross-platform — this is exactly how this project's own real-execution testing was performed, including a test-only substitution of `Mono.Data.Sqlite` for `System.Data.SQLite` that only the test build needs; the shipped code targets the real Windows/.NET Framework provider unmodified.

8.5 Where to Find All of This on Disk

All four implementations, plus the full-length version of every instruction in this chapter, ship together as one self-contained bundle — Static, PHP, JSP, and ASPNET folders side by side, each with its own `VALIDATION_LOG.txt` disclosing exactly how it was tested. A reader who wants to run the code alongside this book should start with that bundle's own `README.md` rather than retype anything from this chapter by hand.

CHAPTER 9

Lessons Beyond the Code

What this case study offers educators, business owners, and students alike.

Chapter 1 named three audiences for this book — students, educators, and business owners or the general public — and the chapters since then have mostly spoken to the first of those three. This chapter steps back and speaks more directly to the other two, drawing out what Profitina Laundry teaches beyond its own source code.

9.1 For Educators: A Case Study Built to Be Adapted, Not Just Read

The specific choices behind Profitina Laundry are, in a real sense, more transferable to another course than the code itself. An educator building their own running case study for a different set of technologies can borrow the same underlying decisions directly: ground the case study in something real rather than inventing it from nothing, so it naturally surfaces genuine complexity (Chapter 3's supervisor relationship, Chapter 4's delete-protection rules) instead of only the complexity a lesson plan anticipated; let each technology's implementation genuinely fit what that block of the course is teaching, rather than forcing every implementation onto one schema from day one, and disclose the resulting divergence plainly rather than hiding it (Chapter 5); and hold every claim in the accompanying material to the same real-execution standard this book applies to itself — a claim that code works should be backed by a log of it actually running, and a bug found along the way should be written up as a lesson, not quietly fixed and forgotten.

> **The most reusable idea in this book**
>
> If one idea from this book is worth carrying into an entirely different course, it is this: a case study grounded in a real, lived business produces better teaching material than one invented purely to fit a syntax lesson, because the real business brings its own genuine edge cases with it — for free, and without anyone having to invent them for effect.

9.2 For Business Owners: What "Digitizing" Actually Involves

A small business owner reading this book with no interest in the code itself can still take something real away from it: an honest, worked example of what turning a paper-and-memory operation into software actually requires, without the sales-pitch framing that usually surrounds that conversation.

- It requires naming, precisely, what the business already tracks informally — Chapter 3's four categories (customers, employees, services, and orders) exist in every laundry business already, whether or not anyone has written them down as a formal list.
- It requires making unwritten rules explicit — a business's staff already know, by habit, that a service someone was already charged for shouldn't just disappear from the price list, or that new customers should go to whoever isn't already busy. Software has to be told these things directly, in a form precise enough for a machine to enforce (Chapter 4's foreign-key rules, Chapter 6's load-balancing logic).

- It requires accepting that the first version will have real bugs, and that finding and fixing them honestly (Chapter 7) is a normal, healthy part of building something real — not a sign that the project went wrong.
- It requires choosing a platform with open eyes about its tradeoffs, the way Chapter 6 and Chapter 8 do for ASP.NET's cross-platform limitations — a real technology decision always has a real cost somewhere, and pretending otherwise catches up with a business eventually.

None of this requires understanding a line of the code in this book. It requires only the willingness to look closely at how the business actually runs today, which is, not coincidentally, exactly what building Profitina Laundry itself required.

9.3 For Students: Reading Real Code, Including Its Flaws

Most code a student encounters in a course is presented already correct, because showing a bug and its fix takes more space than showing only the fix. Profitina Laundry does the opposite on purpose in Chapter 7, and the reason is worth stating directly here: real professional software development is not the tidy, linear process a finished textbook example implies. Bugs are found in code that looked correct on a read-through, sometimes weeks after it was written, and the actual skill being taught by Chapter 7 is not "how ASP.NET ValidationGroups work" so much as "how to isolate an unclear symptom into a minimal reproduction, and how to verify a fix is real rather than merely plausible." That skill transfers to every technology a student will ever work in, long after the specific ASP.NET details are forgotten.

9.4 One General Lesson, Stated Once More

Chapter 1 promised a commitment to honesty — real execution logs, disclosed limitations, bugs named rather than hidden — and every chapter since has tried to keep that promise concretely rather than only as a stated value. That, more than any specific technology or schema decision, is what this book most wants a reader in any of its three audiences to take away: that honest documentation of what actually happened, including the parts that did not go smoothly on the first try, is worth more than a polished account of a process that never quite happened that way.

9.5 What's Left

The final chapter closes the book: where Profitina Laundry stands today, and where it is going next.

CHAPTER 10

Closing: Where Profitina Laundry Goes Next

What remains open, named honestly, and what comes after this book.

Every chapter before this one described Profitina Laundry as it stands today: four working implementations, a canonical data model with real, disclosed design tradeoffs, and at least one real bug found and fixed in the open. This closing chapter looks forward instead — at what remains a deliberate, tracked follow-up, and at where Profitina Laundry is headed next.

10.1 What Is Genuinely Finished

The Static, PHP, JSP, and ASP.NET implementations described in Chapters 5 and 6 are complete and real-execution tested, each against its own real database and real server, with the results disclosed in each track's own `VALIDATION_LOG.txt` rather than only summarized here. Chapter 7's ValidationGroup fix has been verified the same way — against a real SQLite database, not merely reasoned about.

10.2 What Remains a Tracked Follow-Up, Named Honestly

Chapter 4, Section 4.6 and Chapter 5, Section 5.6 already disclosed the most significant open item: the JSP track's flattened `orders` table and the ASP.NET track's canonical `Bill`/`Customer`/`Employee`/`Service` schema model the same business two different ways. Reconciling them properly — updating Chapters 9-11's book text alongside the JSP code, not the code in isolation — is planned as a deliberate next pass over the WebPro course materials, alongside two smaller, similarly honest items: bringing Chapters 14 and 15's book text fully in line with the ASP.NET application as it exists after Chapter 7's fix, and correcting an earlier chapter's mention of XAMPP's setup promise for the PHP track (Chapters 5-6) to match what Chapter 8 of this book actually recommends.

> **Why this is named here instead of quietly fixed first**
>
> A book that only described a finished, reconciled state would be simpler to write, but it would also misrepresent where this project actually stands at the time of publication. Naming an open item honestly — what it is, why it exists, and that it is planned rather than forgotten — is the same commitment to honesty Chapter 1 opened with, applied to the project's own roadmap rather than only to its code.

10.3 Where Profitina Laundry Goes Next

Profitina Laundry's story does not end with the Web Programming course. The same business is planned to appear a fifth time, as the running case study for the Framework-based Programming (FBP) course — a full, independent implementation built the same way every prior track was: real execution, real testing, and its own honestly documented tradeoffs, rather than a copy-and-relabel of any of the four implementations in this book.

Alongside that, this project's broader documentation effort continues past this one book: FBP itself is planned as a complete course, built with the same care as WebPro, and further coursework beyond it remains on this project's roadmap as well.

10.4 A Closing Word

Profitina Laundry began, as Chapter 2 said briefly and deliberately did not elaborate on, as a real business — owned, staffed, and run by real people making real day-to-day decisions. What this book has tried to do across ten chapters is honor that origin by taking the software built around it just as seriously: modeled carefully, tested for real rather than assumed, and documented with its rough edges named rather than smoothed over. If a student, an educator, or a business owner closes this book having learned one thing from it, it is hopefully this: that a small, real business, looked at closely and honestly, has more to teach than most textbook examples ever bother to invent.