Contents

List of Figures

1 Introduction to MVC

1.1 Historical Background

Born 1979
Xerox PARC
Smalltalk-76

In 1979, within the innovative walls of Xerox PARC, Trygve Reenskaug introduced the Model–View–Controller (MVC) architectural pattern to the Smalltalk-76 programming environment. At that time, computing was on the verge of a revolution. Graphical user interfaces (GUIs) were evolving from plain, static text menus into vibrant, interactive experiences—think early drawing tools, windowed desktops, and the very first spreadsheet applications.

This shift came with a challenge: how do you build rich, interactive software that remains

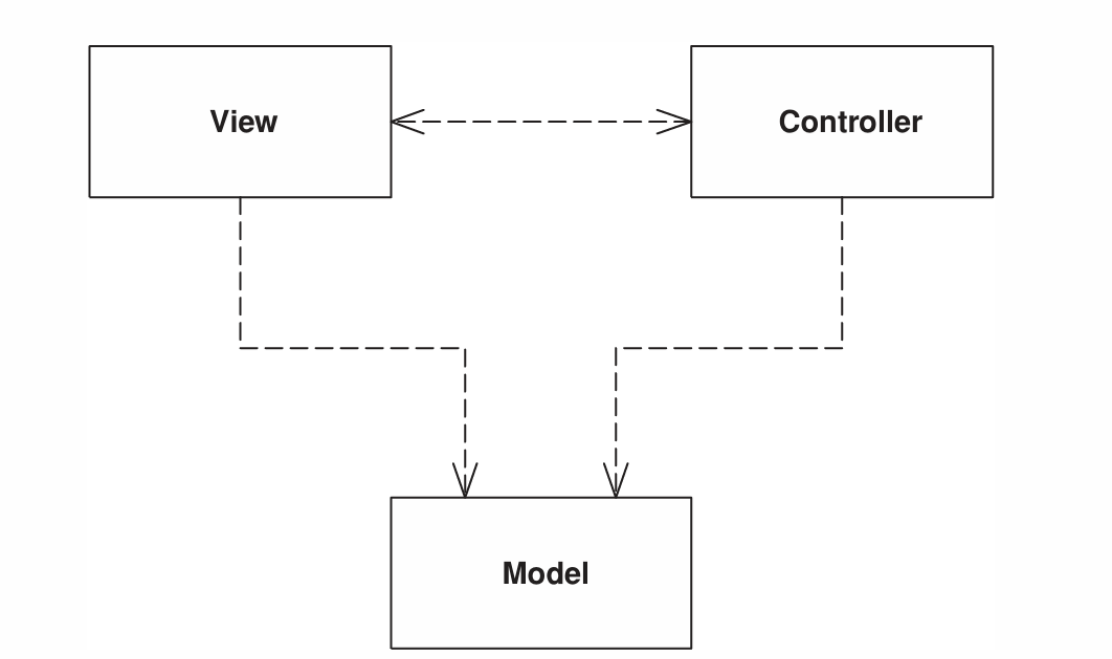

Figure 1.1: *The Model-View-Controller (MVC) pattern separates an application into three interconnected components: the Model (data), the View (UI), and the Controller (logic).*

maintainable, *modular*, and *scalable*? Reenskaug's answer was MVC—a simple yet powerful

idea that divides an application into three clear responsibilities: the **Model**, the **View**, and the **Controller**. This separation not only encouraged code reuse and modularity but also gave development teams a way to collaborate without tripping over each other's work.

> **Tip**
>
> **Why GUIs suddenly got complicated?** Before the late 1970s, user interfaces were almost entirely text-based. Developers built programs around keyboard input and fixed screens. Then came WYSIWYG editors, overlapping windows, and mouse-driven interactions—each introducing new dimensions of complexity:
>
> - Multiple input types arriving at unpredictable times.
> - Continuous updates to the application state.
> - Redraw logic for dynamic, layered visual elements.
>
> In such an environment, *monolithic code quickly became brittle, error-prone, and painful to extend*. MVC's separation of concerns meant you could change a View without rewriting business logic, tweak the Controller without touching data storage, and evolve the Model without breaking the UI. This is the secret sauce behind its decades-long relevance.

By decoupling the internal representation of data from its presentation and interaction logic, MVC empowered developers to think in terms of *collaborating roles* rather than tangled lines of code. The Model manages data and rules, the View handles presentation, and the Controller orchestrates user input and application flow. Together, they form a clear communication triad—minimizing side effects and keeping complexity under control.

> **Note**
>
> **Note:** Whenever you find yourself mixing database calls with UI logic in the same file—stop. That's your cue to revisit MVC's principles. Separation today means fewer headaches tomorrow.

1.1.1 Early Adoption in Desktop Applications

1980s
Desktop GUIs
Smalltalk-80

In its first decade, MVC took root in desktop **GUI systems**—notably Smalltalk-80 applications and early Macintosh software. The division of responsibilities was crisp and practical: the **Model** held the application state and business rules, the **View** rendered visual components (forms, lists, charts), and the **Controller** interpreted keystrokes and mouse events, coordinating updates among the other two.

One compelling advantage was the ability to attach *multiple* Views to the *same* Model—for example, a tabular grid and a chart visualizing the same dataset. Updating the Model would

automatically propagate to all linked Views, eliminating duplicated rendering logic and reducing the risk of subtle state drift.

> **Note**
>
> **Why MVC caught on (then—and still does)?**
>
> - **Parallelism with safety:** Teams could iterate on UI designs (Views) while another group refined business logic (Model), without constant merge conflicts.
> - **Reuse without regret:** The same Model powered multiple visualizations, boosting consistency and reducing bugs caused by copy-pasted logic.
> - **Lower cognitive load:** Clear boundaries meant developers reasoned locally; "What changes where?" had a predictable answer.

Practically speaking, early desktop adopters discovered a durable workflow: *Controllers translate intent*, *Models change truth*, and *Views reflect truth*. That simple triad scaled from toy demos to full-fledged productivity apps—and it remains a mental model you can rely on today.

1.1.2 Evolution Towards Web Frameworks (2000s)

2000s
Web era
HTTP stateless

As the web surged in the late 1990s and early 2000s, MVC crossed from desktop to browser. The move was not a simple port: the *stateless* nature of HTTP, a request/response lifecycle, and the need to scale across servers forced a reinterpretation of roles. Controllers became HTTP request handlers; Views became server-rendered templates; Models connected to databases through object–relational mappers (ORMs). Frameworks such as **Struts**, **Ruby on Rails**, and **ASP.NET MVC** codified these ideas and popularized an opinionated, productive workflow.

> **Note**
>
> Key adaptations in the web era
>
> - **Routing:** Declarative tables map URLs (and HTTP verbs) to Controller actions.
> - **Templating:** Server-side View engines (e.g., JSP, ERB, ASPX/Razor) compose layouts, partials, and helpers.
> - **Persistence:** ORMs (e.g., Hibernate, Active Record, Entity Framework) abstract SQL and enforce domain boundaries.

Web MVC refined responsibilities around the request pipeline. **Controllers** parse parameters, coordinate use cases, and return *representations* (HTML, JSON, files). **Views** focus on presentation concerns: layout, partial reuse, and cache hints. **Models** encapsulate domain rules and data access policies, separating invariants from infrastructure. This clear split improved testability (mockable Controllers and repositories), supported *convention over configuration*, and reduced boilerplate in everyday CRUD flows.

Statelessness also introduced new patterns at scale: session management via cookies or stores; idempotency and safe verbs; cross-cutting concerns handled by filters/middleware (authentication, authorization, caching, CSRF protection, validation). Resource-oriented routing and RESTful conventions aligned URLs with domain nouns, turning Controllers into thin, intention-revealing endpoints rather than monolithic "page scripts."

Two architectural lineages emerged in practice. The first—often called *Model 2* or *request-driven MVC*—funnels every request through a front controller, then dispatches to actions and Views. The second, popularized by Rails, leans hard on folder conventions, naming, and generators to streamline the same idea. Different flavors, same benefits: separation of concerns, parallel development, and simpler reasoning about change.

> **Note**
>
> Adopt *Post/Redirect/Get (PRG)* for mutating requests. Controllers handle the POST (changing Model state), issue a redirect, and Views render the final state on GET. This avoids duplicate form submissions, makes refreshes safe, and clarifies where each concern lives.

1.1.3 MVC in the Era of Single-Page Applications (SPAs)

2010s+
Client-side JavaScript

In the 2010s, the browser became a first-class application platform. MVC ideas migrated client-side as frameworks such as **Backbone.js**, **AngularJS**, **React** (with *Flux*/*Redux* patterns), and **Vue** popularized componentized UIs, client-side routing, and asynchronous data flows. In this setting, the browser hosts the **View** and most "Controller-like" logic; servers expose JSON/-GraphQL APIs; and application state is managed in memory on the client (often centralized in a store).

> **Note**
>
> Key shifts in the SPA era
>
> - **Client-side routing:** URLs map to components/views without full page reloads.
> - **State management:** From ad-hoc mutable state to predictable stores (Flux/Redux, Vuex, NgRx) as a *single source of truth*.
> - **Async-first data:** Fetching via XHR/fetch/GraphQL; caching and revalidation (e.g., React Query/TanStack Query).
> - **Component model:** Views expressed as components with explicit inputs/outputs; "Controller" concerns live in event handlers, hooks, services.

Two notable innovations shaped how "Views" evolve on the client:

- **Two-way data binding** (e.g., AngularJS): changes in inputs update the Model, and Model changes reflect in the View automatically—excellent for forms and prototypes.
- **Virtual DOM rendering** (e.g., React): a diffing step computes minimal DOM updates, avoiding manual manipulation and improving perceived performance on complex UIs.

While early SPAs pushed most work to the client, the pendulum swung toward *hybrid rendering*: **SSR/SSG + hydration** (e.g., Next.js, Nuxt) to deliver fast first paint, SEO-friendly markup, and then "hydrate" into an interactive SPA. More recently, *islands architecture* and *partial hydration* reduce client JavaScript while preserving interactivity where needed.

> **Tip**
>
> Prefer unidirectional flow at scale Two-way binding accelerates small apps, but large SPAs benefit from *unidirectional data flow* (Flux/Redux): actions → reducers → store → view. This improves traceability (time-travel debugging), testability, and performance predictability in complex state graphs.

Beyond state and rendering, production SPAs standardize cross-cutting concerns: code-splitting and tree-shaking for bundle size, client-side caching and pagination, optimistic updates, accessibility, i18n, and offline support via Service Workers. The MVC spirit remains: *separate concerns, make state transitions explicit, and let views be a pure reflection of state.*

> **Warning**
>
> **Over-binding:** Excessive watchers in two-way systems can degrade performance and obscure causality. **Over-centralization:** Forcing all state into a global store leads to "action fatigue." Keep ephemeral UI state local to components; reserve the store for *shared, long-lived, or cross-route* state.

1.1.4 Current Trends and Alternatives (MVVM, MVP, Flux, Clean Architecture)

2015+
Declarative UIs
Unidirectional flow

Today, MVC remains a foundational pattern—often blended with or complemented by *Model–View–ViewModel (MVVM)*, *Model–View–Presenter (MVP)*, unidirectional data-flow patterns like *Flux/Redux*, and layered approaches such as *Clean Architecture*. Its enduring relevance comes from a simple truth: **clear separation of concerns** (data, presentation, orchestration) is the shortest path to maintainable, testable, and scalable systems.

MVVM emerged prominently with Microsoft's WPF/Silverlight toolchains, placing state and UI behavior in a *ViewModel* that exposes observable properties and commands. Views bind declaratively; business rules live outside the UI layer. **MVP**, with roots in Taligent and early Java/.NET UI stacks, keeps the View *passive* while a *Presenter* handles interaction logic and updates the

View explicitly—great for unit-testing UI behavior without a renderer.

On the web, large SPAs embraced **Flux/Redux**, which enforce a *single source of truth* and *unidirectional* data flow (actions → reducers → store → view). Predictability, time-travel debugging, and explicit transitions shine at scale. Meanwhile, **Clean Architecture** separates enterprise concerns into concentric layers (Domain, Application/Use Cases, Interface/Adapters, Infrastructure) and dictates *inward* dependencies. MVC (or MVVM/MVP) often lives in the outer presentation layer, while domain rules remain framework-agnostic at the core.

These approaches are not mutually exclusive. In production systems, you often see:

- MVC or MVVM in the *presentation layer* for composition and rendering.
- Flux-like *state management* for predictable UI transitions at scale.
- Clean Architecture to *isolate domain rules* and keep frameworks replaceable.

> **Note**
>
> Decision guide (choose by constraints)
>
> - **Rapid UI with data binding:** Prefer *MVVM* (forms-heavy apps, desktop/mobile, strong binding support).
> - **Test-first UI behavior:** Use *MVP* (passive Views, explicit updates, presenter unit tests).
> - **Complex SPA state at scale:** Adopt *Flux/Redux* (unidirectional flow, traceability, devtools).
> - **Long-lived core logic:** Enforce *Clean Architecture* (domain/use-case centric, framework-agnostic).

Trendlines. Declarative UIs (React, SwiftUI, Jetpack Compose), server-assisted rendering (SSR/SSG + hydration, islands/partial hydration), MVU/TEA-style loops (Elm) and reactive streams (Rx) all advance the same ideal: *make state transitions explicit, keep views as pure functions of state, and push side effects to the edges*. MVC remains relevant because it teaches that discipline—no matter how the tooling evolves.

> **Note**
>
> Combine patterns intentionally: keep *Controllers/Presenters* thin, push *business rules* into use cases/domain services, and centralize only the *shared, long-lived* state. Let ephemeral UI state stay local to components to avoid "global-store bloat."

1.2 Core Principles of MVC

Three Pillars
Model
View
Controller

This section turns slogans into substance. We will pin down what each layer *owns*, what it *never* does, and how the three cooperate without stepping on each other's toes. Where helpful, we'll flag tips, notes, and warnings—and use short, concrete sketches so the ideas stick.

1.2.1 Model

Model
Data + Rules
Authoritative Source

Definition. The **Model** holds the application's state and business rules. It is the authoritative source of truth for facts such as totals, status, and whether an operation is allowed. When a real-world concept changes (e.g., an order gets paid), that change is expressed through the Model.

1.2.1.1 What the Model does

- Stores and updates state (entities/aggregates and their relationships).
- Enforces rules/invariants (*quantity* > 0, *a closed ticket cannot be reopened by guests*).
- Exposes intention-revealing behavior (e.g., `AddLine`, `Pay`, `Close`) instead of raw property setters.

What the Model does *not* do

- Render UI or format output.
- Handle requests, routing, or transport protocols.
- Contain view-specific tweaks (*badge color*, *page title*); those belong in the View.

Why this matters. When rules live in the Model, you change them once and every Controller/View that uses the Model inherits the correct behavior—preventing duplication and drift.

Listing 1.1: Order Domain Model (PHP, CodeIgniter 4-compatible POPO)

```php
<?php
declare(strict_types=1);

final class Order
{
    /** @var OrderLine[] */
    private array $lines = [];
    private string $status = 'draft'; // draft|paid|closed

    public function addLine(int $productId, int $qty, int $priceCents):
        void
    {
        if ($qty <= 0) {
            throw new DomainException('Quantity must be positive.');
        }
        $this->lines[] = new OrderLine($productId, $qty, $priceCents);
    }

```

```php
    public function totalCents(): int
    {
        $sum = 0;
        foreach ($this->lines as $l) {
            $sum += $l->subtotal();
        }
        return $sum;
    }

    public function pay(string $paymentRef): void
    {
        if ($this->status !== 'draft') {
            throw new DomainException('Already finalized.');
        }
        if ($this->totalCents() === 0) {
            throw new DomainException('Nothing to pay.');
        }
        $this->status = 'paid';
        // domain event: OrderPaid($this, $paymentRef) ...
    }

    public function status(): string
    {
        return $this->status;
    }
}

final class OrderLine
{
    public function __construct(
        public int $productId,
        public int $qty,
        public int $priceCents
    ) {
        if ($qty <= 0) {
            throw new DomainException('Quantity must be positive.');
        }
    }

    public function subtotal(): int
    {
        return $this->qty * $this->priceCents;
    }
}
```

> **Note**
>
> Keep rules close to data Put invariants and operations where the data lives. Validation scattered in Controllers or Views is easy to bypass and hard to test.

1.2.2 View

View
Presentation
No decisions
Render only

Definition. The **View** turns Model data into a presentation: HTML, JSON, PDF, a chart, or a screen. Its job is to *show* the outcome, not decide it.

1.2.2.1 What the View does

- Renders data it receives (a Model, DTO, or ViewModel).
- Applies presentation logic only (formatting dates, pluralization, choosing an icon for a status).

1.2.2.2 What the View does *not* do

- Make business decisions (eligibility, totals, permissions).
- Change the Model directly (it may trigger input events, but does not mutate state on its own).

Sketch (CodeIgniter View): formatting without business rules.

Listing 1.2: Orders View (CodeIgniter 4)

```
<?php
// app/Views/orders/show.php
declare(strict_types=1);

/** @var string $number */
/** @var string $statusCss */
/** @var string $statusText */
/** @var float  $total */          // e.g., 1234.56
/** @var string $currencyCode */   // e.g., 'USD', 'EUR'

helper(['number']); // number_to_currency()

?>
<h2>Order <?= esc($number) ?></h2>

<p>Status:
  <span class="badge <?= esc($statusCss) ?>">
    <?= esc($statusText) ?>
  </span>
</p>

```

```
22 <p>Total: <?= esc(number_to_currency($total, $currencyCode ?? 'USD')) ?></
       p>
```

> **Note**
>
> Views own semantics (headings, labels, roles), alt text, and localized formatting. Keep these concerns out of the Model.

1.2.3 Controller

Controller
Coordinator
Thin layer
Input handler

Definition. The **Controller** coordinates. It processes user input (form posts, button clicks), invokes Model operations, and chooses a View/representation to return. On the web, Controllers are request handlers mapping routes to actions.

1.2.3.1 A well-designed Controller

- Stays thin: delegates business work to the Model or domain services.
- Handles input validation, authorization, and cross-cutting concerns (via filters/middleware where possible).
- Decides the *representation* (HTML, JSON, file) and orchestrates flow (e.g., PRG after a POST).

> **Note**
>
> After a state-changing POST, redirect to a GET. This avoids duplicate submissions on refresh and gives users a stable, bookmarkable URL.

1.3 Communication Flow in MVC

1.3.1 Request–Response Lifecycle Diagram

A typical web request proceeds as follows (front controller pattern):

1. **Routing** selects a Controller action based on URL and HTTP verb.
2. **Model binding** maps inputs (route, query, body) to parameters/DTOs.
3. **Guards** run: authentication, authorization, validation (filters/middleware).
4. **Application logic**: the Controller invokes Model/domain services.
5. **Persistence**: repositories/ORM commit changes (if any).
6. **Result**: the Controller returns a View (HTML) or a serialized body (JSON), or issues a redirect (PRG).
7. **Client** renders the response; follow-up GET fetches idempotent resources.

Tip

Common pitfalls to avoid

- Fat Controllers performing business logic (hard to test, duplicated rules).
- Views computing totals or permissions (logic leaks).
- Skipping PRG, leading to accidental re-posts on refresh.

1.3.2 Data Binding Between Model and View

Data binding connects the Model (or a ViewModel/DTO) to the View. There are two broad strategies:

- **One-way binding** (server-rendered MVC): the Controller passes a Model/ViewModel; the View renders it. Updates occur on the server after form posts or API calls.
- **Two-way or reactive binding** (client frameworks): the View subscribes to state changes and updates automatically; user edits propagate back to state (e.g., Angular's two-way binding, `INotifyPropertyChanged` in MVVM).

Guidelines.

- Prefer one-way flows for predictability in complex apps; localize two-way binding to form widgets.
- Map domain Models to *ViewModels/DTOs* to avoid leaking domain semantics into Views.
- Validate at boundaries: client-side for UX, server-side for correctness.

1.3.3 Observer/Publisher–Subscriber Patterns in MVC

MVC often relies on notification patterns to keep Views in sync without tight coupling.

- **Observer** (pull/push): Views (or Presenters/ViewModels) subscribe to Model changes; the Model raises events or notifies observers.
- **Domain events**: Models publish meaningful events (e.g., *OrderPaid*); listeners handle side effects (email, projections).
- **Reactive streams**: state changes represented as streams; Views compose subscriptions.

Note

Push vs. Pull Push-based notifications reduce polling and keep UIs fresh, but excessive listeners can create hidden couplings. Prefer event names that reflect business meaning (*OrderPaid*) rather than implementation details.

1.4 Advantages of MVC

Why MVC?
Reduced coupling
Clear intent
Better maintainability

The Model–View–Controller (MVC) pattern is widely used because it organizes software into well-defined roles that reduce coupling and clarify intent. This section explains *why* those benefits matter in practice and what they look like on real projects.

1.4.1 Separation of Concerns

SoC
Single responsibility
Localized changes

What it means. MVC assigns distinct responsibilities:

- **Model** — owns state and business rules (invariants, calculations, decisions).
- **View** — renders information (HTML, JSON, PDFs, charts) and handles presentation-only logic (formatting, i18n, layout).
- **Controller** — coordinates a request: parses inputs, invokes Model operations, selects a View.

Why it matters. When each concern has a single place to live:

- Changes stay *localized*. A pricing rule change is made once in the Model, not in every template and endpoint.
- Reasoning becomes simpler. You know where to look for a bug: presentation issues in Views, orchestration issues in Controllers, domain issues in Models.
- Reviews are more effective. Code diffs touch fewer files and fewer layers.

In practice. Teams formalize the handoff between layers (e.g., well-named Model methods and small data structures passed to Views), which prevents accidental leakage of business logic into templates.

> **Tip**
>
> Name Model methods as verbs (`approve()`, `pay()`, `close()`) to keep rules inside the Model and out of Controllers/Views.

1.4.2 Parallel Development by Multiple Teams

Team Scale
Independent work
Clear contracts
Less conflicts

What it means. With clear boundaries, different roles can advance independently: UI engineers focus on Views, backend engineers evolve Models, and platform engineers maintain Controllers/routing.

Why it matters.

- **Throughput.** Work streams proceed in parallel with fewer merge conflicts.
- **Specialization.** Designers and accessibility experts improve Views without touching business rules; domain experts refine Models without impacting rendering.
- **Stable contracts.** Well-defined interfaces between layers become *contracts* that teams can mock while waiting on upstream work.

In practice. Controllers expose small, predictable inputs/outputs; Views consume only the data they need. Contract tests (or snapshots for Views) ensure each side remains compatible as the other changes.

> **Note**
>
> A thin Controller and a passive View make excellent seams for mocking, which lets teams develop and test against each other's interfaces asynchronously.

1.4.3 Improved Testability and Maintainability

Testing
Isolated units
Fast feedback
Clear ownership

What it means. Because responsibilities are isolated, you can test each part *in isolation*:

- **Model tests** verify business rules (totals, eligibility, state transitions) without a web server or templates.
- **Controller tests** check routing, input handling, and response codes without rendering full pages.
- **View tests** validate presentation (formatting, conditional badges, accessibility hints) using fixture data.

Why it matters.

- **Fast feedback.** Most regressions are caught by small, fast unit tests.
- **Safer changes.** Clear ownership reduces unintended side effects across the codebase.
- **Lower cognitive load.** Each test suite focuses on one kind of correctness.

In practice. Keep domain decisions out of Controllers/Views; expose them via Model methods. Controllers return meaningful status codes (`201`, `404`, `409`) that are trivial to assert. Views read precomputed facts and avoid recomputation.

> **Warning**
>
> "Fat Controllers" and "smart templates" undermine testability. If you find yourself duplicating rules in multiple Controllers or Views, move that logic into the Model and test it once.

1.4.4 Scalability in Large Applications

Scale
Modular growth
Independent optimization
Easier refactors

What it means. MVC's modular structure helps applications grow in both *size* (more features) and *complexity* (richer rules, more screens) without collapsing into a monolith.

Why it matters.

- **Feature isolation.** New features arrive as new Controllers, Views, and Model behaviors with minimal cross-impact.

- **Performance options.** You can optimize each layer independently (e.g., cache at the View, batch operations in the Model, add pagination in Controllers).
- **Easier refactors.** You can change rendering strategies (SSR ↔ client components) or adjust routing without rewriting business rules.

In practice. Mature codebases group related Controllers/Views around a Model area (orders, billing, identity). Teams can split work by module, scale review ownership, and roll out changes incrementally.

> **Note**
>
> Before adding a feature, ask:
>
> 1. Which **Model** behavior or rule changes?
> 2. Which **Controller** routes handle the interaction?
> 3. Which **Views** need to render new states?
>
> If you can answer each in one sentence, your MVC boundaries are probably healthy.

1.5 Common Misconceptions & Pitfalls

Pitfalls
Fat Controllers
Anemic Models
Logic in Views
Over-engineering

Despite its longevity and clarity, Model–View–Controller (MVC) is frequently misapplied. This section clarifies four recurrent mistakes, explains *why* they are harmful, and offers practical refactoring guidance.

1.5.1 Fat Controllers and How to Refactor Them

Fat Controller
Business logic in wrong place
Hard to test

Problem. A *fat controller* accumulates business decisions, calculations, and ad hoc validations. This violates separation of concerns and turns a boundary object into the place where rules live.
Symptoms.

- Long controller actions with nested conditionals and branching.
- Duplication of rules across multiple actions or controllers.
- Difficulty unit-testing rules without spinning up a web stack.

Why it hurts. Controllers change frequently as endpoints evolve; embedding rules here breeds inconsistency, slows testing, and couples business logic to transport details.
Refactoring checklist.

1. Identify decision points (eligibility checks, totals, transitions) and extract them into **Model** methods with verb names (e.g., `approve()`, `cancel()`, `pay()`).
2. Keep the controller as *orchestrator*: parse input → call Model → select View/response.
3. Collapse repeated controller branches into a single Model call that returns a clear outcome (value, state change, or domain error).

> **Tip**
>
> A healthy controller reads like a short story: *parse input* → *invoke a domain verb* → *render*. If you see calculations or policy decisions, move them into the Model.

1.5.2 Anemic Models vs. Rich Domain Models

Anemic Model
Data without behavior
Rules scattered

Problem. An *anemic model* holds data but no behavior: setters everywhere, rules elsewhere. This pushes the system toward procedural code and scatters invariants.
Symptoms.

- Public setters on core entities; little or no validation inside the Model.
- Controllers or helpers recompute totals, statuses, and eligibility in multiple places.
- Fragile tests that must stand up infrastructure to exercise business rules.

Why it hurts. Rules duplicate, drift, and become hard to find. Small UI changes risk breaking core behavior because the truth is not centralized.
Refactoring checklist.

1. Replace setters with **intent-revealing methods** (`addLine`, `reopen`, `applyDiscount`) that enforce invariants internally.
2. Embed **invariants** in the Model (fail fast if violated); expose **queries** like `isPaid()` for read-side clarity.
3. Keep presentation-only concerns (labels, colors, formatting) out of the Model.

> **Note**
>
> A *rich* Model encapsulates both state and behavior. Controllers call verbs; Views call queries. Tests target the Model directly with no UI or routing in the loop.

1.5.3 Business Logic Leakage into Views

Logic in Views
Business rules in templates
Hard to maintain

Problem. Templates or UI components implement business rules (pricing, eligibility, status transitions) because "it was convenient" during rendering.
Symptoms.

- Conditional blocks replicate rules (e.g., price calculations) across multiple templates.
- Designers can unknowingly change behavior by editing presentation code.
- Inconsistent results between different screens for the same scenario.

Why it hurts. Views change rapidly; mixing rules with presentation guarantees duplication and divergence. It also undermines testability.
Refactoring checklist.

1. Move decisions into **Model queries and commands** (e.g., `isOverdue()`, `canBeCancelled()`).

2. Pass the *result* of those decisions to the View and keep template logic presentational only (formatting, layout, i18n).
3. Remove any remaining computations from templates; verify with View tests against simple, stable inputs.

> **Warning**
>
> If a template change can "break a rule," the rule is in the wrong place. Views should *show* outcomes decided elsewhere, not *decide* them.

1.5.4 Over-Engineering Small Applications with MVC

Over-engineering
Too much complexity for simple cases

Problem. Applying full MVC structure to trivial use cases introduces ceremony without benefit.
When to keep it simple.

- A handful of screens with minimal rules and no reuse pressure.
- One-off utilities or prototypes with short lifespans.
- Static content with light form handling.

Why it matters. Architecture is a trade-off: indirection, interfaces, and layers cost time and attention. Use them where they *pay for themselves*.
Pragmatic guidance.

1. Start with a *thin* MVC slice if the framework encourages it, but resist premature layering.
2. As rules or teams grow, promote duplicated checks into Model methods and introduce clear boundaries.
3. Let structure *evolve* with complexity; do not front-load abstractions you do not need.

> **Tip**
>
> Ask: (1) How many distinct rules exist? (2) How often will they change? (3) How many places must render the same facts? If answers trend upward, lean into MVC's separation; if not, keep the path short.

1.6 MVC in Different Languages and Frameworks

MVC travels well, but every ecosystem speaks it with a local accent. The mappings below clarify *what counts as Model/View/Controller* in each stack, the default tooling (templating, routing, ORMs), and the habits that keep projects maintainable.

1.6.1 MVC in PHP (Symfony, Laravel)

In PHP, MVC matured around batteries-included frameworks.

Laravel. *Controllers* live in `app/Http/Controllers`, *Views* are Blade templates in `resources/views`, and *Models* are Eloquent classes (*Active Record* style) in `app/Models`. Routing is declarative (`routes/web.php`, `routes/api.php`); form requests centralize validation; migrations seed and evolve schema.

Symfony. Symfony uses a *front controller* (`public/index.php`) with attribute/annotation-based routing in `src/Controller`. Twig handles *Views*. *Models* are typically Doctrine entities (*Data Mapper*) in `src/Entity`, with repositories in `src/Repository`.

> **Tip**
>
> Prefer *form requests* (Laravel) or *form types/validators* (Symfony) for validation; pass *ViewModels/DTOs* to templates. Keep database work in repositories/services—not in Controllers.

1.6.2 MVC in Python (Django)

Django brands its variant **MTV**:

- **Model** (Django ORM) → tables, relations, migrations, signals.
- **Template** (Jinja-like engine) → the *View* in classic MVC terms.
- **View** (function or class-based) → acts like the *Controller*; it receives the request, talks to the Model, returns a response.

URLs map to Views through `urls.py`; middleware handles cross-cutting concerns; the admin gives instant CRUD for data exploration.

> **Note**
>
> Favor *class-based views* for reuse; extract domain logic to *services* or *domain models* to avoid "fat views." Use *forms/serializers* to validate input at the boundary.

1.6.3 MVC in Ruby (Ruby on Rails)

Rails is the canonical "*convention over configuration*" MVC:

- **Model** = *ActiveRecord* objects (`app/models`) with validations, associations, callbacks.
- **View** = ERB/HAML templates (`app/views`) plus helpers/partials/layouts.
- **Controller** = `app/controllers`, RESTful actions generated by scaffolds; filters for cross-cutting concerns.

Routing is resource-oriented; migrations are first-class; environments and credentials are standardized.

> **Tip**
>
> Overuse of callbacks can hide business rules. Prefer *service objects*, *form objects*, and *domain events* for complex workflows.

1.6.4 MVC in Java (Spring MVC)

Spring MVC centers on the *DispatcherServlet* (front controller). Controllers are annotated (`@Controller`, `@RestController`, `@RequestMapping`); *Views* are typically Thymeleaf/Freemarker; *Models* are POJOs plus Spring Data repositories (`@Entity`, JPA/Hibernate). DI (`@Service`, `@Component`), validation (Bean Validation/JSR 380), and *ControllerAdvice* for cross-cutting concerns round out the stack.

> **Tip**
>
> Return *DTOs* from controllers; translate to domain types in services. Encapsulate transactions at the service boundary (`@Transactional`); keep controllers thin and idempotent for GET.

1.6.5 MVC in .NET (ASP.NET MVC & ASP.NET Core MVC)

ASP.NET (Core) MVC uses attribute routing, *Model Binding*, and Razor for Views. *Controllers* return `IActionResult` (HTML/JSON/file); filters handle auth, validation, caching; DI is built-in. *Models* are domain entities plus EF Core repositories; *ViewModels* keep Views presentation-only.

> **Tip**
>
> Use *PRG* after POST; prefer *ViewModels/DTOs* to prevent over-posting; apply `[FromBody]` and validation attributes at the boundary. Keep business invariants inside domain entities/services.

1.6.6 MVC Adaptations in JavaScript Frameworks (Angular, Backbone.js)

Client-side frameworks adapted MVC to the browser's evented runtime.

- **Angular** leans MVVM/MV with components, templates, services, and RxJS. Two-way binding is available; unidirectional flows via `OnPush` and reactive patterns scale better.
- **Backbone.js** offers minimalist *Models* (events, REST sync), *Views* (DOM glue), and a *Router*; templating is library-agnostic.

> **Tip**
>
> Keep ephemeral UI state local to components; reserve centralized stores for *shared, cross-route, or long-lived* state.

Ecosystem	Model	View	Controller	Routing/Templating
Laravel	Eloquent (AR)	Blade	Controllers	`routes/*`, Blade
Symfony	Doctrine (DM)	Twig	Controllers	Attributes, Twig
Django	ORM (Model)	Template	View (Controller)	`urls.py`, Templates
Rails	ActiveRecord (AR)	ERB/HAML	Controllers	REST routes, ERB
Spring	JPA/Hibernate	Thymeleaf	`@Controller`	DispatcherServlet/Thymeleaf
ASP.NET	EF Core	Razor	`Controller`	Attr. routing/Razor
Angular	Services/State	Templates	Components (MV)	Router/Templates
Backbone	Model + Sync	Templating lib	View + Router	Router/Underscore

1.7 Modern Adaptations & Critiques (with a CodeIgniter/PHP Lens)

MVC is the outer shell in many systems. Inside (or alongside) it, teams often apply MVVM, MVP, Flux/Redux, or domain-centric architectures (Clean/Hexagonal). Below we translate these patterns to a *CodeIgniter 4 + PHP* context with small, practical sketches.

1.7.1 MVVM (Model–View–ViewModel)

MVVM inserts a *ViewModel* that exposes observable state and commands. On the web, PHP doesn't do runtime observation in templates; instead, treat the ViewModel as a *presentation adapter* (DTO/array) that the controller passes to the view. If you need live reactivity, pair CI4 with a client MVVM (Vue/Alpine/htmx) or a small event stream.

- **Strengths:** excellent separation of presentation state; makes templates simple and testable.
- **Practices (PHP):** build a ViewModel DTO; map from domain in one place (service/presenter); no business rules in the view.
- **Watch-outs:** "Massive ViewModel" anti-pattern—split by feature (e.g., `OrderDetailsVm`, `OrderSummaryVm`).

Listing 1.3: MVVM-style ViewModel adapter in CI4

```php
<?php
// app/ViewModels/OrderDetailsVm.php
declare(strict_types=1);
namespace App\ViewModels;

final class OrderDetailsVm
```

```
{
    public function __construct(
        public string $number,
        public string $statusText,
        public string $statusCss,
        public float  $total,
        public string $currencyCode = 'USD',
        public array  $lines = []
    ) {}

    public function toArray(): array
    {
        return get_object_vars($this);
    }
}

// app/Services/OrderQueries.php
namespace App\Services;

use App\Domain\OrderRepositoryInterface;
use App\ViewModels\OrderDetailsVm;

final class OrderQueries
{
    public function __construct(private OrderRepositoryInterface $repo) {}

    public function detailsVm(int $id): OrderDetailsVm
    {
        $o = $this->repo->get($id);
        return new OrderDetailsVm(
            number: 'ORD-'.$o->id(),
            statusText: ucfirst($o->status()),
            statusCss: $o->status()==='paid'?'badge-success':'badge-
                secondary',
            total: $o->totalCents()/100,
            lines: $o->lines()
        );
    }
}

// app/Controllers/Orders.php
namespace App\Controllers;

final class Orders extends BaseController
{
    private \App\Services\OrderQueries $queries;
    public function __construct() { $this->queries = service('orderQueries
        '); }

    public function show(int $id)
    {
```

```
56          $vm = $this->queries->detailsVm($id);
57          return view('orders/show', $vm->toArray()); // one-way binding
58      }
59  }
```

> **Tip**
>
> Treat ViewModels as *state adapters*. Map domain → ViewModel in a single service/presenter; pass arrays to the view. Keep domain objects out of templates.

1.7.2 MVP (Model–View–Presenter)

MVP keeps the View *passive*. The Presenter handles input, talks to the Model, and *pushes* a ready-to-render model into the template. In PHP this maps naturally: a Presenter composes an array; the View echoes it.

- **Variants:** Passive View (no logic in templates) vs. Supervising Controller (some binding).
- **Use cases:** complex formatting/multi-step screens you want to unit test without hitting HTTP.
- **Trade-offs:** more classes; mitigate with small, feature-scoped presenters.

Listing 1.4: MVP-style Presenter feeding a passive CI4 view

```
1  <?php
2  // app/Presentation/OrderPresenter.php
3  namespace App\Presentation;
4
5  use App\Domain\Order;
6
7  final class OrderPresenter
8  {
9      public function present(Order $o): array
10     {
11         return [
12             'number'     => 'ORD-'.$o->id(),
13             'statusCss'  => $o->status()==='paid'?'badge-success':'badge-
                   secondary',
14             'statusText' => ucfirst($o->status()),
15             'total'      => number_format($o->totalCents()/100, 2),
16         ];
17     }
18 }
19
20 // app/Controllers/Orders.php (snippet)
21 public function show(int $id)
22 {
23     $order = $this->orders->get($id);           // domain
```

```
    $data  = (new \App\Presentation\OrderPresenter())->present($order);
    return view('orders/show', $data);          // passive view
}
```

> **Tip**
>
> Define a tiny presenter per feature; unit test it with stubbed domain objects. Keep the presenter stateless and pure.

1.7.3 Flux/Redux Architectures in React

Flux/Redux enforces *unidirectional* flow: Actions trigger state changes, which then update the view.

1.8 Best Practices for Implementing MVC (CodeIgniter 4, PHP)

The practices below keep your CI4 apps predictable, testable, and easy to evolve. Each subsection includes a succinct rationale, a checklist, and compact PHP sketches.

1.8.1 Keeping Controllers Slim

Goal: Controllers coordinate input → use case/service → response. No business rules here.

Checklist.

- Validate request data early; map input to simple DTO/array.
- Delegate domain work to a service (`service('orderService')`).
- Use *PRG* (Post/Redirect/Get) for state-changing actions.
- Return representations (views/JSON/files), not domain entities.

Listing 1.5: Orders Controller (CodeIgniter 4)

```
<?php
// app/Controllers/Orders.php
declare(strict_types=1);

namespace App\Controllers;

use CodeIgniter\HTTP\RedirectResponse;

final class Orders extends BaseController
{
    private \App\Services\OrderService $orders;

    public function __construct()
    {
```

```
        $this->orders = service('orderService'); // swappable in tests
    }

    public function pay(int $id): RedirectResponse
    {
        $rules = ['payment_id' => 'required|string'];
        if (! $this->validate($rules)) {
            return redirect()->back()->withInput()
                ->with('errors', $this->validator->getErrors());
        }

        $this->orders->pay($id, (string) $this->request->getPost('
            payment_id'));
        return redirect()->to(route_to('orders_show', $id)); // PRG
    }

    public function show(int $id)
    {
        $vm = $this->orders->detailsVm($id); // ViewModel/DTO
        return view('orders/show', $vm);
    }
}
```

> **Tip**
>
> If an action grows past ~20–30 lines or branches on business rules, extract the logic to a service/use case and keep the controller orchestration-only.

1.8.2 Ensuring Models Contain Business Logic

Goal: Put invariants and operations where the data lives. Keep domain logic in plain PHP (*POPOs*); let CI `Model` act as a repository (persistence adapter).

Checklist.

- Express rules with intention-revealing methods (`pay()`, `addLine()`), not raw setters.
- Keep domain pure; hide DB concerns behind a repository.
- Emit domain events (optional) for cross-cutting reactions.

Listing 1.6: Domain & Repository (CodeIgniter 4)

```
<?php
// app/Domain/Order.php
namespace App\Domain;

final class Order
{
```

```
    private string $status = 'draft'; // draft|paid|closed
    /** @var array<int,array{productId:int,qty:int,priceCents:int}> */
    private array $lines = [];
    public function __construct(private int $id) {}

    public function addLine(int $pid, int $qty, int $priceCents): void {
        if ($qty <= 0) throw new \DomainException('Qty must be positive.');
        $this->lines[] = ['productId'=>$pid,'qty'=>$qty,'priceCents'=>$priceCents];
    }
    public function totalCents(): int {
        return array_sum(array_map(fn($l)=>$l['qty']*$l['priceCents'], $this->lines));
    }
    public function pay(string $paymentId): void {
        if ($this->status !== 'draft') throw new \DomainException('Finalized.');
        if ($this->totalCents() <= 0) throw new \DomainException('Nothing to pay.');
        $this->status = 'paid';
    }
    public function id(): int { return $this->id; }
    public function status(): string { return $this->status; }
    public function lines(): array { return $this->lines; }
}

<?php
// app/Repositories/OrderRepository.php
namespace App\Repositories;

use App\Domain\Order;
use App\Domain\OrderRepositoryInterface;
use App\Models\OrderModel;

final class OrderRepository implements OrderRepositoryInterface
{
    public function __construct(private OrderModel $orders) {}

    public function get(int $id): Order {
        $row = $this->orders->find($id);
        if (! $row) throw new \RuntimeException('Order not found');
        $order = new Order($id);
        // hydrate status/lines here...
        $ref = new \ReflectionProperty($order, 'status');
        $ref->setAccessible(true);
        $ref->setValue($order, $row['status']);
        return $order;
    }

    public function save(Order $order): void {
```

```
53          $this->orders->save([
54              'id' => $order->id(),
55              'status' => $order->status(),
56              'total_cents' => $order->totalCents(),
57          ]);
58          // persist lines as needed...
59      }
60  }
```

> **Note**
>
> Even if your CI `Model` is *Active Record*-ish, keep business rules inside the domain object. This preserves testability and prevents logic leakage into controllers/views.

1.8.3 Leveraging View Templates Effectively

Goal: Views render; they do not decide. Templates should contain presentation logic only.

Checklist.

- Pass ViewModels/DTOs tailored for the template; avoid exposing entities.
- Centralize formatting (currency, dates, badges) in helpers/partials.
- Enforce accessibility (labels, roles, alt text) and i18n in views.

Listing 1.7: View Template (CodeIgniter 4)

```
1  <!-- app/Views/orders/show.php -->
2  <h2>Order <?= esc($id) ?></h2>
3  <p>Status:
4    <span class="badge <?= esc($statusCss) ?>">
5      <?= esc($statusText) ?>
6    </span>
7  </p>
8  <p>Total: <?= esc($total) ?></p>
9  <?= view('orders/_lines', ['lines' => $lines]) ?>
```

> **Tip**
>
> If you see eligibility checks or permission logic in templates, surface them as boolean flags on the ViewModel and compute them in the domain/service layer.

1.8.4 Applying Dependency Injection in MVC

Goal: Make collaborators explicit and swappable. CI4's `Services` provide simple DI for controllers/services.

Checklist.

- Wire dependencies in `app/Config/Services.php`.
- Resolve with `service('name')` in runtime code; inject mocks in tests.
- Keep transactions at service boundaries; avoid service location in domain.

Listing 1.8: Services Layer (CodeIgniter 4)

```php
<?php
// app/Config/Services.php
namespace Config;

use CodeIgniter\Config\BaseService;
use App\Services\OrderService;
use App\Repositories\OrderRepository;
use App\Models\OrderModel;

class Services extends BaseService
{
    public static function orderRepository(bool $getShared=true) {
        if ($getShared) return static::getSharedInstance('orderRepository'
            );
        return new OrderRepository(new OrderModel());
    }
    public static function orderService(bool $getShared=true) {
        if ($getShared) return static::getSharedInstance('orderService');
        return new OrderService(static::orderRepository(false));
    }
}
```

> **Warning**
>
> Avoid pulling the framework container from deep inside domain objects. Keep the domain pure; perform composition at the edges (controller/service).

1.8.5 Writing Unit and Integration Tests for MVC Applications

Goal: Test behaviors at the right level: domain first (fast), web boundary second (feature), and DB as needed (integration).

Strategy.

- **Unit (fast):** domain entities/services; no framework, no DB.
- **Feature:** HTTP layer (routes, filters, responses) with `FeatureTestTrait`.
- **Database:** use `DatabaseTestTrait` or Testcontainers; migrate/seed per test.

Listing 1.9: Domain Unit Test

```php
public function test_pay_requires_positive_total(): void
{
    $o = new \App\Domain\Order(1);
    $this->expectException(\DomainException::class);
    $o->pay('p-1');
}

use CodeIgniter\Test\FeatureTestTrait;
use Config\Services;

final class OrdersFeatureTest extends \CodeIgniter\Test\CIUnitTestCase
{
    use FeatureTestTrait;

    public function test_pay_redirects_on_success(): void
    {
        $fake = $this->createMock(\App\Services\OrderService::class);
        $fake->expects($this->once())->method('pay')->with(42, 'p-1');

        Services::injectMock('orderService', $fake); // swap service

        $result = $this->post('orders/42/pay', ['payment_id' => 'p-1']);
        $result->assertStatus(302);
        $result->assertRedirectTo(route_to('orders_show', 42));
    }
}
```

Listing 1.10: Database Integration Test

```php
use CodeIgniter\Test\DatabaseTestTrait;

final class OrdersDbTest extends \CodeIgniter\Test\CIUnitTestCase
{
    use DatabaseTestTrait;

    protected $migrate = true;
    protected $refresh = true;

    public function test_show_renders_order(): void
    {
        $this->seeInDatabase('orders', ['id' => 1, 'status' => 'paid']);
        $res = $this->get('orders/1');
        $res->assertOK()->assertSee('Order 1');
    }
}
```

Tip

- Mock *boundaries* (services, repos), not value objects.
- Assert observable behavior (status, redirects, HTML fragments), not private state.
- Keep tests independent; reset services with `\Config\Services::reset()` between runs if needed.

1.9 Summary and Further Reading

1.9.1 Key Takeaways

- **Separation of concerns is the north star.** Keep *domain rules* in Models (plain PHP where possible), *presentation* in Views, and *coordination* in Controllers.
- **Prefer intention-revealing APIs.** Domain methods like `pay()`, `addLine()`, `close()` communicate business intent better than property setters.
- **Make Controllers thin.** Validate input, delegate to a use case/service, choose a response. Apply *PRG* (Post/Redirect/Get) after mutations.
- **Shape a View Model for each template.** Pass arrays/DTOs that are presentation-ready; avoid leaking entities into views.
- **Keep the domain framework-agnostic.** Treat framework Models as persistence adapters; keep business logic in POPOs to improve testability and portability.
- **Use DI and composition at the edges.** Wire dependencies in a single place (`Services.php` in CI4); inject mocks in tests.
- **Test at the right levels.** Fast unit tests for domain rules, feature tests for the HTTP layer, and integration tests for persistence.
- **Scale state with discipline.** For SPAs, prefer unidirectional data flow; on the server, keep JSON contracts stable and versioned.
- **Reach beyond MVC when constraints demand it.** Use MVVM/MVP for presentation complexity; Clean/Hexagonal for deep business rules.

Tip

Quick checklist for your next MVC feature

1. Write/adjust a domain method that captures the rule.
2. Add a use case/service that calls the domain and saves via a repository.
3. Map domain → ViewModel/DTO in one place.
4. Keep the controller to validation + delegation + PRG.
5. Add a domain unit test, a feature test for the route, and (if needed) an integration test for persistence.

1.9.2 Recommended Books and Articles

- *Patterns of Enterprise Application Architecture* — Martin Fowler.
- *Domain-Driven Design: Tackling Complexity in the Heart of Software* — Eric Evans.
- *Implementing Domain-Driven Design* — Vaughn Vernon.
- *Clean Architecture* — Robert C. Martin.
- *Growing Object-Oriented Software, Guided by Tests* — Freeman & Pryce.
- *Refactoring* (2nd ed.) — Martin Fowler.
- *Designing Data-Intensive Applications* — Martin Kleppmann.
- *A Personal History of Model–View–Controller* — Larry Tesler, *ACM Queue*.
- *Domain-Driven Design in PHP* — Carlos Buenosvinos, Christian Soronellas, Keyvan Akbary.
- *Laravel: Up & Running* — Matt Stauffer (useful even if you are on CI, for PHP patterns and testing culture).

> **Tip**
>
> Skim architecture books for *boundaries and dependencies*; dive deep into DDD for modeling rules; use testing books to make changes safe. Treat framework books as *how* and architecture texts as *why*.

1.9.3 Official Documentation References for Popular MVC Frameworks

(Enable `\usepackage{hyperref}` to make the links clickable.)

- **CodeIgniter 4 (PHP):** User Guide — Controllers, Routing, Models, Validation, Testing, Services.
- **Laravel (PHP):** Documentation — Routing, Controllers, Blade, Eloquent, Testing.
- **Symfony (PHP):** Docs — Controller, Routing, Twig, Validator, Doctrine.
- **Django (Python, MTV):** Docs — Models, Templates, Views, URL routing, Testing.
- **Ruby on Rails (Ruby):** Guides — MVC, Active Record, Action Controller, Action View.
- **Spring MVC (Java):** Spring Web MVC Reference.
- **ASP.NET Core MVC (C#):** ASP.NET Core MVC Docs.

Tip: Bookmark each framework's sections on *testing*, *validation*, and *dependency injection*. Mastery of these three areas buys you the most maintainability per hour invested.