Advanced Web Programming

SIB245007 | D-IV Sistem Informasi Bisnis

Meeting 2: **HTTP Protocol and the MVC Pattern**

Request/Response Cycle & Web Application Architecture

What You'll Learn

1. Explaining the HTTP **request/response** cycle (method, status code, header) and mapping it onto routing code
2. Comparing the **MVC**, **MVVM**, and **Clean Architecture** patterns, and explaining how Laravel implements MVC
3. Recognizing the roles of **route**, **controller**, and **middleware**, including modern routing patterns: **route parameters**, **named routes**, **route groups**, and **resource routing**
4. Explaining how to organize controllers: **resource controllers**, **single-action controllers**, and the *thin controller* principle

> This slide covers concepts. The practical steps for writing routes, controllers, and middleware are covered separately outside this slide.

Part 1

The Post Office Analogy

A letter

- Comes in through the intake counter
- The clerk reads the address & type of mail
- Forwarded to the right department
- The clerk doesn't open the envelope or decide the reply's content

A request to a web application

- Comes in through one door
- Sorted by its address & type
- Forwarded to the right handler for processing
- The sorter doesn't decide the response's content

> If the sorting goes wrong (e.g. a delete-data request gets forwarded without checking whether the sender is an admin), the application loses control over who's allowed to change what.

Anatomy of a Request

> **Request:** what a browser sends to a server, carrying a method, an address (URL), headers, and sometimes data (a form, JSON).

> **Route:** a rule mapping one combination of method and URL to the code that will handle it.

- Every request always carries a **method** stating the request's intent
- It's this method that determines which route matches, not the URL alone

HTTP Methods

Method	Meaning	Typical Use
`GET`	Requesting data, **without changing** anything server-side	Listing data
`POST`	Submitting new data	Saving new data
`PATCH`	Updating part of existing data	Updating part of a record
`DELETE`	Removing data	Deleting a record

Other methods worth knowing: **`PUT`** (replacing all of the data at once), **`HEAD`** (like `GET` but only asking for headers, no body), **`OPTIONS`** (asking which methods the server allows).

> Full list of methods: `developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods`

Method Ground Rules: Safe & Idempotent

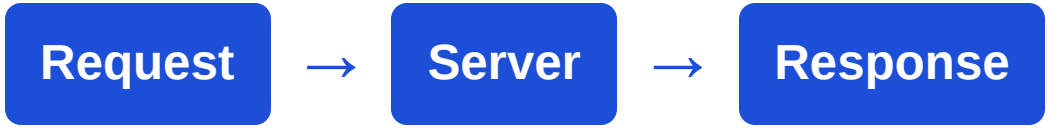
> **Safe:** methods that don't change anything server-side: `GET`, `HEAD`. **Idempotent:** methods that produce the same result no matter how many times they're sent: `GET`, `PUT`, `DELETE`; `POST` is not.

- Browsers & servers assume this rule is followed: refresh, the back button, and caching all depend on it
- `POST` isn't idempotent, which is why the save-then-redirect pattern on the next slide is needed

> Strict rule: `GET` must never change data server-side. Breaking this rule makes application behavior unpredictable, e.g. refreshing a page accidentally deleting data.

Anatomy of a Response

Response: what a server sends back to a browser, carrying a status code, headers, and a body (HTML, JSON, a redirect).



A response always carries a **status code**, a three-digit number summarizing the outcome before a single byte of the body gets read.

The Five Status Code Classes

Hundreds of status codes are grouped by their first digit: you only need to memorize the five classes, not every code.

Class	Meaning	The Gist
`1xx`	Informational	"Received, still processing", rarely seen directly
`2xx`	Success	The request was processed successfully
`3xx`	Redirection	The browser is told to go to another address
`4xx`	Client Error	A mistake on the sender's side (wrong address, invalid data)
`5xx`	Server Error	A mistake on the server's side

> Full list: `developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status`

Status Codes on a Web Application

Code	Meaning	Typical Use
200	OK	A list page renders successfully
302	Redirect	After a form saves, the browser is sent to the detail page
404	Not Found	Opening a detail page for an ID that doesn't exist
422	Unprocessable Entity	A form is submitted with invalid data
500	Server Error	An unhandled error on the server side

- Notice the pattern: 2xx = success, 3xx = address change, 4xx = sender-side mistake, 5xx = server-side mistake

The 302 Pattern: Save-then-Redirect

POST /articles → **302 + Location** → **GET /articles/{id}**

> After `POST /articles` successfully saves data, the server doesn't send HTML back right away: it sends a 302 response with a `Location` header telling the browser to request another address instead.

- This pattern repeats across almost every data-writing feature
- Stops a user from hitting **refresh** and accidentally submitting the same data twice

Headers: Metadata Outside the Body

> **Header:** metadata that accompanies a request or response, outside its main body.

Request headers (browser → server)

- `Content-Type`: format of the data being sent
- `Accept`: the response format wanted
- `Authorization`: sender's identity token
- `Cookie`: session data sent back

Response headers (server → browser)

- `Content-Type`: format of the response body
- `Location`: the redirect target address
- `Set-Cookie`: the server hands off session data
- `Cache-Control`: whether the response may be cached

Full list of headers: `developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers`

Seeing HTTP Directly

- Modern browsers provide **DevTools → the Network tab** to see every request-response that happens
- Each row shows: method, address, status code, and response time
- The most direct way to prove that "behind every click is a real HTTP request"

> You'll put this straight into practice: opening DevTools and watching real requests from routes you build yourself.

Part 2

The Problem MVC Solves

- Just accepting requests and sending responses isn't enough to keep code clean as an application grows
- Without a firm separation of responsibilities, code that "sorts requests", "fetches data", and "builds the display" ends up mixed in one file
- The **MVC** (Model-View-Controller) pattern answers this by drawing a firm line between three responsibilities

Model, View, and Controller

> **MVC:** the Model owns data, the View owns presentation, the Controller owns the request flow between them.

Back to the post-office analogy

- Controller = the sorting clerk
- Model = the filed records holding the actual data
- View = the reply form handed to the sender

Practical rule for where code goes

- Touches data → Model
- Handles request flow → Controller
- Renders output → View

> The framework doesn't enforce this: a query inside a view still runs. This pattern is a discipline the code's author has to maintain.

MVC in Laravel's Folder Structure

Folder	MVC Role
`routes/web.php`	Controller: URL registration
`app/Http/Controllers/`	Controller: request-handling classes
`app/Models/`	Model: data representation & business rules
`resources/views/`	View: what the user sees

> This folder structure isn't an accident: it embodies the same MVC pattern as the diagram earlier.

MVVM

> **MVVM (Model-View-ViewModel):** inserts a ViewModel between Model and View; the ViewModel holds view state and keeps it automatically synced to the View through two-way binding.

- Popular in **reactive** interface applications: the display keeps changing without a page reload
- More relevant to frontend frameworks (e.g. Vue) than to server-rendered applications in general

Clean Architecture

Clean Architecture: isolates core business rules (use cases) entirely from whatever framework is in use, so they can be tested, and even moved to a different framework, without being touched.

- Benefit: business rules can be tested and moved without depending on the framework
- Cost: an extra abstraction layer that needs maintaining

For a small-to-medium scale application, isolation this strict adds abstraction that isn't yet worth its cost. Laravel's built-in MVC is already clean enough.

Comparing the Three Patterns

Pattern	Main Separation	Typical Use
MVC	Model, View, Controller	Laravel (built-in)
MVVM	Model, View, ViewModel	Applications with two-way binding between view and state
Clean Architecture	Use-case layer isolated from the framework	Large systems with heavy business logic

Part 3

From URLs to clean controllers

Route, Controller, and Middleware

> **Controller:** a class holding request-handling methods, invoked by whichever route matches.

> **Middleware:** a layer that inspects or transforms a request before it reaches a controller, for example checking whether the user is logged in.

- The hands-on practicum will apply all three to the Simple POS case study

Route Parameters: Dynamic Addresses

> **Route parameter:** a part of the address written in curly braces, e.g. `{id}`, whose value gets filled in from the real URL and passed to the controller.

```
Route::get('/articles/{id}', [ArticleController::class, 'show']);
```

- `/articles/1`, `/articles/2`, `/articles/9999`: one route serves them all
- The `{id}` value is received by the `show` method as an argument
- A missing ID → **404** (see the status code table back in Part 1)

Named Routes: Addresses Can Change, Names Don't

> **Named route:** a unique label attached to a route via `->name()`, so the rest of the application refers to that name instead of the raw address.

```
Route::get('/login', [LoginController::class, 'create'])
    ->name('login.create');
Route::post('/login', [LoginController::class, 'store'])
    ->name('login.store');
```

- `GET /login` and `POST /login` are **two different routes** even though the address is the same, distinguished by their method
- Links/redirects are written as `route('login.create')`. Address changes from `/login` to `/signin`? No caller needs editing
- Naming convention: `source.action`, e.g. `articles.index`, `articles.store`, etc.

Route Groups: Shared Rules

Route group: wraps several routes so they share the same middleware, address prefix, or name prefix, written once, applied to all of them.

```
Route::middleware('auth')->group(function () {
    Route::get('/dashboard', [DashboardController::class, 'index'])
        ->name('dashboard');
    Route::get('/settings', [SettingController::class, 'edit'])
        ->name('settings.edit');
});
```

A new route that forgets to be wrapped in the right middleware group is an easy-to-miss security hole: a delete-data route meant only for admins becomes accessible to anyone who knows the address. Always check a new route's position before considering it done.

Resource Controllers: Seven Conventional Actions

Any CRUD feature always needs the same seven actions. Laravel standardizes them into a naming convention for controller methods.

Action	Method	URL	Task
`index`	GET	`/articles`	List all articles
`create`	GET	`/articles/create`	Add-article form
`store`	POST	`/articles`	Save a new article
`show`	GET	`/articles/{id}`	A single article's detail
`edit`	GET	`/articles/{id}/edit`	Edit-article form
`update`	PATCH	`/articles/{id}`	Save changes
`destroy`	DELETE	`/articles/{id}`	Delete an article

`Route::resource`: Seven Routes, One Line

```php
Route::resource('articles', ArticleController::class);
```

- This one line registers all 7 routes from the previous slide at once, complete with named routes (`articles.index`, `articles.show`, etc.)
- Only need some of them? `->only(['index', 'show'])`
- The same convention across every feature means anyone on the team instantly knows where an action lives

> The `php artisan route:list` command shows a table of every registered route (method, URL, name, and middleware) without opening route files one by one.

Single-Action Controller

> **Single-action controller:** a controller with one `__invoke()` method for one single task, used when an action doesn't make sense folded into the seven resource actions.

```php
Route::get('/articles/{id}/pdf', ExportArticlePdfController::class);
```

- Exporting an article as a PDF isn't `show`, isn't `update`: it's a standalone action
- Its route points straight at the class, with no method name
- A sign you need one: an action keeps getting "forced" into a resource method that doesn't fit

The Thin Controller Principle

> **Thin controller:** a controller only handles flow: accepting a request, calling the right party, choosing a response. Business rules live in the Model (or a service layer), not the controller.

Fat controller (avoid)

- Calculate totals, validate input, send notifications, format output: all in one method
- Hard to test, hard to reuse

Thin controller (the goal)

- `store` only validates input, hands the calculation off to the Model, then redirects
- The business logic can be reused from anywhere

> Remember the rule from Part 2: touches data → Model, handles flow → Controller. A fat controller is a slow-motion violation of MVC: each line feels small, until one day your `store` method is 200 lines long.

Summary

- A request carries a **method** (`GET` / `POST` / `PATCH` / `DELETE` + `PUT` / `HEAD` / `OPTIONS`); a response carries a **status code** split into five classes (`1xx` – `5xx`); **headers** carry metadata in both directions
- **MVC** separates Model (data), View (presentation), Controller (flow); **MVVM** suits reactive interfaces; **Clean Architecture** suits large systems. Laravel maps MVC directly onto its folder structure
- Modern routing: **route parameters** (`{id}`), **named routes** (`->name()`), **route groups** (shared middleware/prefix), and **`Route::resource`** which registers seven actions at once
- Controller organization: **resource controllers** for CRUD, **single-action controllers** for standalone actions, and the **thin controller** principle: flow in the controller, business rules in the Model

References & Discussion

Official Laravel documentation · MDN Web Docs (developer.mozilla.org) · Fielding's (2000) dissertation on REST

Full code: `github.com/se-polinema/simple-pos`

Next meeting: Frontend & Templating (Blade, Tailwind, Alpine)