# Advanced Web Programming

## SIB245007 | D-IV Sistem Informasi Bisnis

Meeting 3: **Frontend & Templating (Blade, Tailwind, Alpine)**

The Frontend Landscape, Server-Side Templating, and Light Interactivity

# What You'll Learn

1. Compare the **MPA** and **SPA** patterns, and explain where Blade and Alpine.js sit as a hybrid approach
2. Understand the **template engine** concept and the **utility-first CSS** approach, and how the two work together to build a page's layout
3. Explain how **Alpine.js** adds client-side interactivity with no page reload, while the server stays the source of truth

> This slide deck covers concepts. Building the layout, the cashier page, and the cart happens in the practicum jobsheet, this time as a group.

# Part 1

# A Glass Display Case vs. an Attentive Clerk

### Glass display case

- Never changes until the owner walks over and rearranges it
- A customer who wants to know how much stock is left has to call over a clerk and wait

### Attentive clerk

- The moment a customer lifts an item, the clerk already knows
- Recalculates the running total on the spot, ready to undo that pick without the customer starting over

> A page built purely with Blade, with no JavaScript touching it, behaves like the glass display case: every time a user adds one item to a cart, the entire page reloads. An application with high interaction volume doesn't have time for dozens of reloads in a single session.

# MPA: Reload the Whole Page

> **Multi-page application (MPA):** every time a user follows a link or submits a form, the browser discards the old HTML, requests a new one, and renders it from scratch.


- Simple to reason about and easy to debug: every request gets back a complete HTML answer
- Feels sluggish for small, frequently repeated interactions, e.g. adding one product to a cart

# SPA: JavaScript Takes Over

> **Single-page application (SPA):** one HTML page loads once at the start, and every change after that is handled by JavaScript in the browser through hidden requests, updating only part of the DOM with no reload.

- React, Vue, and Angular are libraries built specifically around this pattern
- Fits very frequent interaction and complex state
- The cost: build complexity and state split across two places (client and server)

# A Spectrum, Not Two Poles

**Pure MPA** → **Server-rendered + a little JavaScript** → **Pure SPA**

- Most real applications sit in the middle, not at either end
- The ecosystem has names for this middle ground: **Livewire** (Laravel), **Hotwire** (Rails), **htmx**

> In this approach, the server stays the source of truth; JavaScript's only job is speeding up the one part that would otherwise feel slow if it had to reload in full.

# Comparing Three Patterns

| Pattern | Render Source | Fits |
| --- | --- | --- |
| Pure MPA | Server, full HTML per navigation | Pages with infrequent interaction, where SEO matters |
| Pure SPA | Client, JavaScript renders the DOM | Applications with very frequent interaction, complex state |
| Blade + Alpine.js | Server for structure, client for local interaction | Apps with mostly static pages, a small part needing reactivity |

## When Blade + Alpine.js Is Enough

- Pages that rarely change within a single work session (e.g. lists, reports): plain server rendering through Blade is already fast enough
- The part that needs repeated interactivity without a reload (e.g. a shopping cart): Alpine.js fills that gap
- No need to force the whole application into a different architecture just for this one small part

> Moving the entire application to a full SPA just for one small component is an architectural cost that doesn't pay off.

# Part 2

Rendering a page from the server

# The Template Engine Concept: Template + Data → HTML

> **Template engine:** a tool that merges a template containing placeholders with real data, producing a final document ready to display.

**Template (placeholder)** → **+ Data** → **Template Engine** → **Final HTML**

- A simple example: template `<h1>Hello, {{ $name }}</h1>` + data `$name = "Rani"` → result `<h1>Hello, Rani</h1>`
- The same concept appears across languages: Blade (Laravel/PHP), Jinja (Python), EJS (JavaScript), Twig (PHP)

> Blade is one implementation of this concept in the Laravel ecosystem, not the concept itself.

# Blade: Laravel's Templating Engine

> **Blade:** Laravel's built-in templating engine, rendering HTML on the server using directive syntax like `@if` and `@foreach` directly inside `.blade.php` files.

> **Directive:** a Blade instruction starting with `@`, such as `@extends`, `@section`, and `@yield`, compiled by Laravel into plain PHP before it runs.

- Blade isn't a new language, just a concise way to write PHP inside a template

# Layout: Write the Skeleton Once

Nearly every page in a web application shares the same skeleton: a tab title, a navigation menu, and a content area that changes. A Blade layout avoids repeating that skeleton in every file.

```
<!-- resources/views/layouts/app.blade.php -->
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@yield('title', 'App Name')</title>
    @vite(['resources/css/app.css', 'resources/js/app.js'])
</head>
<body>
    <x-nav />
    <main>@yield('content')</main>
</body>
</html>
```

# A Page Fills the Layout's Hole

```
<!-- resources/views/articles/index.blade.php -->
@extends('layouts.app')

@section('title', 'Article List')

@section('content')
    <h1 class="text-lg font-semibold mb-4">Article List</h1>
    <div class="grid grid-cols-3 gap-4">
        @foreach ($articles as $article)
            <div class="border rounded-md p-3">
                <p class="font-medium">{{ $article->title }}</p>
            </div>
        @endforeach
    </div>
@endsection
```

- The page never rewrites `<html>`, `<head>`, or the navigation menu

> Full list of Blade directives: `laravel.com/docs/blade`

# `{{ }}` and `{!! !!}`: Printing Data into HTML

**`{{ }}`:** Blade's syntax for printing a variable or PHP expression's value into HTML, automatically escaping HTML characters.

**`{!! !!}`:** Blade's syntax for printing a variable's value with no escaping at all; its content is printed as-is, as raw HTML.

- Example: `{{ $article->title }}` prints the article's title, automatically safe from a character like `<` that could otherwise be abused to inject foreign markup

Data coming from user input, including an article title typed into a form, must always be printed through `{{ }}`, never `{!! !!}`. The second syntax skips escaping entirely and opens a cross-site scripting hole the moment its content ever came from untrusted input.

- The full security discussion arrives in the validation & security meeting

# Component: A Piece You Reuse

> **Component:** a reusable piece of Blade shared across pages, such as a product card or a button, registered as its own file and invoked through an `<x-component-name>` tag.

- `<x-nav />` in the layout above is a component
- An article card or a button are natural next candidates

# Vite: The Frontend Build Tool

> **Vite:** the frontend build tool Laravel ships with by default, running a dev server with hot reload through `npm run dev` during development, and bundling/minifying every CSS & JavaScript file into production assets through `npm run build`.

**app.css / app.js** → **Vite** → **Tags via @vite**

- `@vite([...])` replaces manual `<script>` / `<link>` tags

## Three Files Behind `@vite`

1. `resources/css/app.css`: the first line is `@import 'tailwindcss';`, no separate `tailwind.config.js` file
2. `vite.config.js`: registers the `laravel()` and `tailwindcss()` plugins
3. `package.json`: lists both packages as `devDependencies`

> All three have been in your project since Meeting 1. This meeting you start actually using them.

## Two Terminals: `artisan serve` + `npm run dev`

- Vite runs as a dev server separate from `php artisan serve`, in a second terminal
- What to expect: `VITE vX.X.X ready` followed by a local address
- Editing `app.css` or a Blade file shows up instantly, with no manual refresh (hot reload)

> If `npm run dev` stops, the `@vite` directive on a page that reloads will fail to find the dev server and throw a `ViteManifestNotFoundException`.

> `composer run dev` runs artisan serve, npm run dev, and other processes together in one terminal. Start with two separate terminals first, so it's clear which process fails.

# Tailwind: Utility-First CSS

- Utility classes attach directly to elements, instead of separate CSS rules in another file
- Example: `class="bg-slate-900 text-white rounded-md px-4 py-2"` for a dark, rounded button with padding
- One class equals one small styling decision, easy to guess and easy to remove

> Full utility-class reference: `tailwindcss.com/docs` (don't memorize it, look it up when you need it)

# Part 3

A shopping cart with no page reload

# Alpine.js: State Inside the Markup

> **Alpine.js:** a lightweight JavaScript library adding state and interactivity directly through HTML attributes ( `x-data` , `x-model` , `@click` ), installed through npm like any other JavaScript dependency.

- Not a mini SPA framework, but a complement to a page Blade already renders
- Any element inside an `x-data` , including its children, can read and change that state through other Alpine attributes

## The Core Mechanism: `x-data`

```
<div x-data="{ cart: [] }">
  <button @click="cart.push({ id: 1, price: 15000 })">
    Add
  </button>
  <span x-text="cart.length"></span> items in cart
</div>
```

- `cart` starts as an empty array, the Add button pushes one new object on every click
- `x-text="cart.length"` updates the number automatically, with no line of code explicitly telling the DOM to refresh
- Alpine watches `cart` for changes and keeps the display in sync on its own

# Alpine Attributes You'll Use

| Attribute | Purpose |
| --- | --- |
| `x-data` | Declares local state |
| `@click` | Runs code when an element is clicked |
| `x-text` | Displays a reactive value |
| `x-for` | Repeats an element for each item |
| `x-model` | Binds an input to state |

> Full attribute list: `alpinejs.dev`

# Installed via npm, Not a CDN

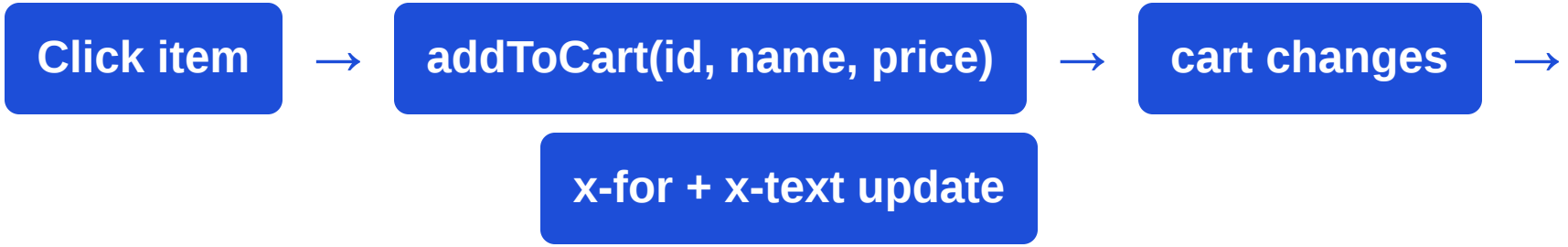
```
npm install alpinejs
```

```
// resources/js/app.js
import Alpine from 'alpinejs';

window.Alpine = Alpine;
Alpine.start();
```

- `app.js` is already loaded by `@vite` in the layout, no new `<script>` tag needed
- The bundler's load order automatically sidesteps the problem the `defer` attribute used to solve on a CDN tag

# Interactivity Pattern: Click to Update State




- `x-data` wraps the relevant list of items, defining `cart`, `addToCart`, and `subtotal()`
- `subtotal()` uses `reduce` to sum up the prices

> Proof there's no reload: the address bar and tab favicon never flicker.

# The Golden Rule: The Server Recalculates

> The subtotal Alpine computes client-side exists purely for display. Every time the form gets submitted, the server recalculates the total from the data in the database, not from whatever number Alpine displayed in the browser. Anything coming from the browser can still be tampered with before it reaches the server.

> A useful rule of thumb: cosmetic logic (showing a subtotal, highlighting a just-added item) is safe to keep in Alpine. Business decisions (the final total, data validity) must be recalculated on the server.

## The Grown-Up Version: `Alpine.data`

- An inline `x-data` object is enough to learn the mechanism
- In the Simple POS case study you build in the jobsheet, the cart registers itself through `Alpine.data('posCart', ...)` in `app.js`, complete with SKU scanning, discounts, and `sessionStorage`
- Same concept, different scale

> Full code: `github.com/se-polinema/simple-pos` , branch `chapter-03`

# Summary

- An MPA reloads the whole page on every navigation, an SPA renders everything client-side, and Blade + Alpine.js take a middle path: the server still renders the structure, Alpine only adds reactivity where it's needed
- A Blade layout through `@extends` / `@section` / `@yield` avoids repeating HTML skeleton code; `@vite` loads Tailwind CSS and JavaScript through Vite, with `npm run dev` giving hot reload
- Tailwind works through utility classes attached to elements; `{{ }}` escapes automatically, `{!! !!}` does not and opens an XSS hole
- Alpine.js is installed through npm; `x-data` declares state right in the markup, `@click` and `x-text` read and change it reactively, and a client-computed subtotal must still be re-verified on the server

# References & Discussion

Official Laravel documentation (Blade, Vite) · Tailwind CSS (tailwindcss.com) · Alpine.js (alpinejs.dev)

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

**Next meeting:** Database Design, Migrations, and Seeding