Advanced Web Programming

SIB245007 | D-IV Sistem Informasi Bisnis

Meeting 1: **Modern Web Architecture and the Laravel Ecosystem**

Semester Learning Plan (RPS) & Meeting 1 Materials

Part 1

Course Information

Study Program	D-IV Sistem Informasi Bisnis
Course Code	SIB245007
Course Name	Advanced Web Programming

> This course uses **Simple POS**, a real point-of-sale cashier app for small businesses that grows commit by commit, as a running case study across the whole semester.

Semester Learning Plan

Meeting	Topic
1	Modern Web Architecture
2	HTTP & MVC Architecture
3	Frontend & Templating
4	Database Design & Migration
5	ORM & Data Relations
6	Input Validation & Security
7	Authentication & Authorization
8	**Midterm**
9	Data Processing & Export

Meeting	Topic
10	API Architecture & Design
11	PBL Project Planning
12	PBL Feature Development
13	PBL Integration & Testing
14	PBL Optimization & Deployment
15	PBL Project Finalization
16	PBL Final Exam Preparation
17	**Final Exam**

Meetings 1–10 build the technical foundation in Laravel; Meetings 11–17 shift it into an independent *Project Based Learning* (PBL) project.

Evaluation Components

Evaluation Basis	Weight
Participatory Activity (Case Method)	0%
Project Outcome (Project Based Learning)	55%
Cognitive – Weekly assignment (increment)	15%
Cognitive – Quiz	0%
Cognitive – Midterm (project progress + presentation)	5%
Cognitive – Final Exam (final project + presentation)	25%
Total	**100%**

> **Weekly assignment**: incremental (*increment*) implementation of the Simple POS application, submitted as evidence of code and documentation per the rubric.

Part 2

What You'll Learn

1. Comparing **monolith**, **microservices**, and **serverless** architectures to understand the basis for choosing a web application's architecture
2. Understanding why **Laravel 13** with **SQLite** as a zero-setup database was chosen for the Simple POS project you'll build across the semester
3. Recognizing the Laravel project folder structure (`routes/`, `app/Http/Controllers/`, `database/migrations/`) as an embodiment of the **MVC** pattern

> This slide covers concepts. Installation steps, project setup, and full hands-on practice are covered separately outside this slide.

What Is Web Architecture?

> **Web architecture:** how an application's layers (interface, business logic, data access) are organized and deployed: as a single unit, or as many separate units.

- Choosing an architecture isn't just a technical decision: it determines how many deploy processes, failure points, and network calls the team has to manage
- A "fancier" architecture isn't automatically better: building a food court for a business that only needs one kitchen just burns effort on connecting plumbing instead of features
- Three styles we'll compare: **monolith**, **microservices**, **serverless**

Monolith

> **Monolith:** An application whose layers (interface, business logic, data access) all run in a single codebase and a single deploy process.

- Adding a feature = adding code to the same project
- Deploying an update = replacing one unit with a new version
- Often mistaken for "messy code"
- A well-structured monolith (e.g. with MVC) stays clean
- Its opposite isn't "modular", it's **"distributed"**

Microservices

> **Microservices:** An architecture that splits an application into independent services, each running and deployed on its own, communicating over a network.

- Each service: different language, different deploy schedule, scales independently
- **The price:** 8 services = 8 deploy processes + 8 failure points + network calls between services
- Worth it for **large teams** running systems at millions-of-users scale
- For a small-scale application: the cost far outweighs the benefit

Analogy: Family Restaurant vs. Food Court

The two architectures we just defined, in an everyday analogy:

Family restaurant (monolith)

- One kitchen, one register, one team
- Everybody knows everything
- Busy? Add a stove, not a new branch

Food court (microservices)

- Each stall: its own kitchen, register, recipe
- Independent units
- One empty stall doesn't affect the others

> An application is built as a **monolith** (the family restaurant) not out of limitation, but because the scale fits: a shop with one or two cashiers doesn't need ten separate services.

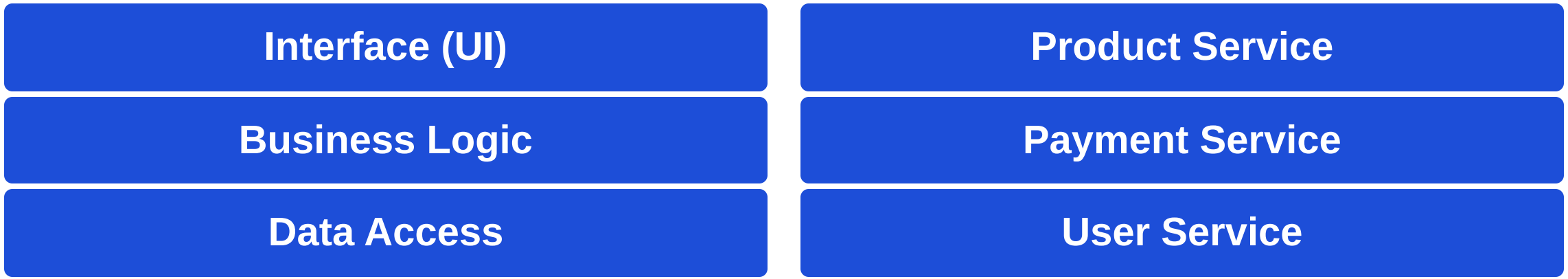
Serverless

> **Serverless:** Code runs as small functions that only execute when triggered by an event, with no server process staying up continuously.

- The name is misleading: the server still exists, it's just not your responsibility
- There's a **cold start** delay when a function hasn't been called in a while
- Billing is per execution, not per hour a server stays on
- Good fit: spiky workloads (e.g. image processing on upload)
- Poor fit: an application that needs a consistent database connection

Comparing the Three Architectures

Architecture	Deployment	Initial Complexity	Best For
Monolith	Single unit	Low	Small-scale apps, MVPs, small teams
Microservices	Many independent units	High	Large-scale systems, large teams
Serverless	Function per event	Medium	Sporadic workloads



Monolith

Microservices

Why Laravel?

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

- One PHP project: routing, authentication, ORM, templating, ready to use
- Compared to plain PHP: consistent MVC structure from the first line of code
- Compared to microservices-first frameworks: stays productive for a team of 1–2 people
- Strong package ecosystem: **Sanctum** (API), **Excel** (import/export), **Cashier** (payments)
- Similar philosophy to Django (Python) & Ruby on Rails: *convention over configuration*

One Entry Point: `public/index.php`

Plain PHP

- URL maps directly to a file
- `/product.php` → runs `product.php`
- Every file is potentially accessible directly via URL

Laravel

- Every request enters through one file: `public/index.php`
- URLs are registered in `routes/web.php`
- Other files (Controller, Model) can't be accessed directly via URL

> A single entry point means every request can be processed uniformly before reaching the application code: the foundation of routing, middleware, and centralized authentication.

One Laravel Request, Start to Finish

> This flow pattern repeats across every Laravel feature, from a simple page to REST API endpoints, all following the same path.

Setting Up the Project: Why SQLite?

- The database is stored in **a single plain file**
- No separate server process to start and configure
- Migrations can run right in the first minute
- Simple POS runs on SQLite even for classroom demos

> Before continuing, make sure these four tools are installed: `php -v` (8.2+), `composer -V`, `node -v`, and `git --version`.

Composer & npm: Dependency Managers

> **Composer:** PHP's dependency manager, reads `composer.json`, downloads packages into `vendor/`, then generates `vendor/autoload.php`.

- Thanks to `autoload.php`, every class is usable right away, no manual `require` / `include` like in plain PHP
- **npm** is its counterpart in the JavaScript world: `package.json` lists packages, `node_modules/` stores them, used by Laravel for Vite & Tailwind
- Both folders (`vendor/`, `node_modules/`) are downloaded output, never committed to Git

`.env` & the Configuration Layer

> **.env:** a configuration file that separates credentials and environment settings from source code.

- Configuration read path: `.env` → the `env()` helper → `config/*.php` → the `config()` helper used by application code
- The app key inside it is used to encrypt sessions and cookies
- This separation allows different configuration per environment (local, staging, production) without changing code

> The `.env` file stores sensitive data and must never be committed. Laravel's default `.gitignore` already excludes it.

Artisan & Migrations

> **Artisan:** Laravel's built-in CLI for everyday development tasks (migrations, seeding, generating boilerplate): a dev tool, not part of the application served to users.

> **Migration:** a PHP file that defines a database schema change in code, so the schema can be rebuilt consistently on any machine.

- Treat migrations like version control for the schema: a new change means a new migration file, never edit an old migration that's already run elsewhere
- The `--seed` option fills tables with realistic sample data for practice and demos

Laravel's Folder Structure = MVC in Practice

Folder	MVC Role	Contents
`routes/web.php`	Controller	URL registration
`app/Http/Controllers/`	Controller	Request-handling classes
`database/migrations/`	Model	Database schema definitions
`resources/views/`	View	Blade files (Meeting 3)
`vendor/`	-	Composer packages, not committed

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

Summary

- **Monolith** unifies every layer in one codebase and one deploy, a fit for small-to-medium scale applications; **microservices** splits it apart at a cost that's only worth it for large systems; **serverless** suits sporadic workloads
- Laravel is chosen as a framework because its structure stays consistent from the start (MVC) and it's productive for small teams; **SQLite** is the database because it's zero-setup: one file, no separate server
- **Composer**/npm manage dependencies; **.env** separates configuration from code; **Artisan** & **migrations** build the database schema programmatically
- Laravel's folder structure embodies the **MVC** pattern, consistently separating the responsibilities of routing, business logic, and presentation

References & Discussion

Official Laravel documentation · PHP Manual

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

Next meeting: HTTP & MVC Architecture