Practicum Jobsheet: Meeting 1

Setting Up a Laravel Project and Simple POS Git Repository

Course	Advanced Web Programming (SIB245007)
Meeting	1 (Week 1)
Duration	2 sessions × 170 minutes
Sub-CPMK	Sub-CPMK 1: Students can understand the basic concepts of web frameworks and apply routing, controllers, and database management in web application development.
Ending Code	`chapter-01` branch at `github.com/se-polinema/simple-pos` (to compare against your own result)

A. Practicum Outcomes

After completing this jobsheet, you'll be able to:

1. Set up a new Laravel 13 project with SQLite as its database, complete with migrations and sample data.
2. Run Laravel's development server and verify that the welcome page displays correctly.
3. Recognize the parts of Laravel's folder structure you'll touch often throughout the semester.
4. Initialize a Git repository and make a first commit following the `increment N` convention.

B. Preparation and Prerequisites

- **Tools**: PHP 8.2 or newer, Composer, Node.js (for `npm`), and Git.
- **Continuing the code**: none, this meeting starts from scratch, the project gets created directly on your computer.
- **Quick check** before starting, run all four one by one in a terminal:

  ```
  php -v
  composer -V
  node -v
  git --version
  ```

  If any command isn't recognized, or your PHP version is below 8.2, install/update it before moving on to Step 1.

C. Work Steps

Step 1: Configure your Git identity (once per computer)

Git stamps every commit with its author's name and email. If the computer you're using has never used Git before, set this up now so Step 7 later doesn't fail partway through.

```
git config --global user.name "Your Name"
git config --global user.email "your-email@example.com"
```

> ✓ **Checkpoint:** the command prints nothing (it succeeds silently). Verify with `git config --global user.name`: it should print the name you just entered.

Step 2: Creating a new Laravel project

`composer create-project` runs Composer, PHP's dependency manager: it reads `composer.json`, downloads every package into a `vendor/` folder, then generates `vendor/autoload.php`, a file that makes every class in the project usable right away without manual `require`/`include` like plain PHP.

```
composer create-project laravel/laravel simple-pos
cd simple-pos
cp .env.example .env
php artisan key:generate
```

> ✓ **Checkpoint:** `composer create-project` prints a list of downloaded packages, ending with a line like `Application ready in simple-pos. You can now start using Composer!`. The `key:generate` command prints `INFO Application key set successfully.`

> ⚠ **If it fails:** `composer create-project` stopping with a message about the PHP version (e.g. `requires php ^8.2`) means you should run `php -v` to confirm the installed version is 8.2 or newer. Laravel 13 can't be installed on an older version. If the process stops from a network timeout, rerun the same command; Composer resumes from the packages it hasn't downloaded yet.

Step 3: Installing frontend dependencies

Besides `composer.json`, a fresh Laravel project also ships a `package.json`: a list of JavaScript dependencies for the frontend toolchain (Vite and Tailwind CSS) that Meeting 3 starts using. `npm install` is the JavaScript-world equivalent of `composer install`: it reads `package.json`, then downloads every package into a `node_modules/` folder.

```
npm install
```

> ✓ **Checkpoint:** the output ends with a line like `added N packages in Ns`, with no `npm error` lines in between.

This step isn't required to run Simple POS today. Laravel's default page still shows up even without `node_modules/`. Running it now avoids a sudden installation delay once Meeting 3 starts using Tailwind and Alpine.js through Vite.

Step 4: Connecting to SQLite and running migrations

This project uses SQLite instead of MySQL/PostgreSQL: the entire database lives in one plain file, with no separate server process to start and give credentials to. Make sure the following line is in your `.env` (Laravel 13's default already sets this, so you're just verifying it, not changing it):

```
# .env
DB_CONNECTION=sqlite
```

Create the database file, then run every migration along with the sample data:

```
touch database/database.sqlite
php artisan migrate:fresh --seed
```

> ✓ **Checkpoint:** a list of migrations prints one by one, each ending with the word `DONE`, with no error lines, followed by a message that the seeder finished running.

> ⚠ **If it fails:** a `database file does not exist` message means the `touch` step above wasn't run, or ran at the wrong location (it must be exactly `database/database.sqlite`). A `could not find driver` message means the `pdo_sqlite` PHP extension isn't enabled: check with `php -m | grep sqlite`, enable it in `php.ini` if it doesn't show up, then try again.

Step 5: Running the development server

```
php artisan serve
```

> ✓ **Checkpoint:** a line `INFO  Server running on [http://127.0.0.1:8000]` prints, and the terminal stays open waiting for requests. Open `http://127.0.0.1:8000` in your browser. Laravel's welcome page, not an error message, is the sign every previous step succeeded. Press `Ctrl+C` to stop the server once you're done verifying.

> ⚠ **If it fails:** a message about port 8000 already being in use means another process (maybe a previous `php artisan serve` session that wasn't stopped) is using that port; run `php artisan serve --port=8001` and open `http://127.0.0.1:8001` instead.

Step 6: Getting to know the project structure

Before writing your first feature code in Meeting 2, get familiar with the parts of Laravel's folder structure you'll touch often throughout the semester. Open the following four folders in your code editor, then fill in the table below without writing any code.

1. `routes/` (specifically `routes/web.php`): where every URL the application's users can visit gets registered.
2. `app/Http/Controllers/`: where classes that process each request (accepting input, calling a model, choosing a view) will grow one by one starting Meeting 2.
3. `database/migrations/`: where database schema changes are defined as code, instead of being clicked through in a separate database application.
4. `vendor/`: holds every third-party Composer package, compare it against the three folders above.

> ✅ **Checkpoint (fill in this table in your report):**

Question	Your Answer
How many default migration files are in `database/migrations/`?	
Name 2 tables those default migrations create	
Is it okay to edit `vendor/` by hand? Why?	
Does `vendor/` get committed to Git? Why?	

Step 7: Initializing Git and making the `increment 1` commit

`composer create-project` doesn't set up a Git repository automatically, so this step is done by hand from inside the `simple-pos/` folder.

```
git init
```

> ✅ **Checkpoint:** a message `Initialized empty Git repository in .../simple-pos/.git/`.

Stage every project file, then make the first commit:

```
git add .
git commit -m "increment 1: proyek Laravel kosong"
```

> ✅ **Checkpoint:** a summary of how many files were committed, ending with a line like `N files changed, M insertions(+)`. Notice that `vendor/`, `node_modules/`, and `.env` aren't listed. Laravel's default `.gitignore` already excludes all three.

Verify the commit was saved:

```
git log --oneline
```

> ✅ **Checkpoint:** one line with a short hash followed by the message `increment 1: proyek Laravel kosong`, exactly as you just wrote it.

> ⚠ **If it fails:** if `git commit` refuses with a message asking for `user.name`/`user.email`, go back to Step 1, run both `git config --global` commands there, then retry `git commit` here.

D. Tasks and Deliverables

Submit the following in the format your teaching assistant/instructor requests:

- A screenshot of Laravel's welcome page at `http://127.0.0.1:8000`.
- The output of `git log --oneline` showing your `increment 1` commit.
- The Step 6 table (folder structure exploration).
- **Independent task:** explain in your own words, in 3-5 sentences, why a food stall with a single cashier is better suited to a monolith architecture than microservices, while a national e-commerce platform with millions of users often uses microservices.

E. Grading Rubric

Component	Weight	Full Marks (100%)	Minimum Marks
Work steps completed	40%	The project runs, all of Steps 1-7 done	Server runs, some steps done
Checkpoints verified	30%	Welcome page screenshot + `git log` + exploration table complete and correct	Some checkpoints proven
Independent task	20%	The monolith vs. microservices explanation is accurate and self-written (not copied from the material)	An answer exists but is incomplete
Commit hygiene	10%	The commit message is exactly `increment 1: proyek Laravel kosong`, `vendor//node_modules//.env` not committed	A commit exists, but the message or contents are messy