# Practicum Jobsheet: Meeting 2

## Routes, Controllers, and Observing HTTP Directly

| | |
|---|---|
| **Course** | Advanced Web Programming (SIB245007) |
| **Meeting** | 2 (Week 2) |
| **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. |
| **Starting Code** | `chapter-01` branch at `github.com/se-polinema/simple-pos` |
| **Ending Code** | `chapter-02` branch at `github.com/se-polinema/simple-pos` |

## A. Practicum Outcomes

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

1. Observe the HTTP request/response cycle directly through browser DevTools and `curl`.
2. Create a controller with Artisan and register it through a route.
3. Tell apart two routes with the same address but different methods.
4. Wrap a group of routes with middleware and explain its effect.

## B. Preparation and Prerequisites

- **Tools**: same as Meeting 1 (PHP 8.2+, Composer, Node.js, Git), plus `curl` (already available by default on macOS/Linux; on Windows it's available through modern PowerShell or Git Bash).
- **Continuing the code**: continue your own `simple-pos` project from Meeting 1. If you've fallen behind or your project has problems, start from this meeting's starting code:

  ```
  git clone -b chapter-01 https://github.com/se-polinema/simple-pos.git
  cd simple-pos
  composer install
  npm install
  cp .env.example .env
  php artisan key:generate
  touch database/database.sqlite
  php artisan migrate:fresh --seed
  ```

- **Quick check** before starting:

  ```
  git log --oneline
  ```

  It should show at least one `increment 1: proyek Laravel kosong` line. Also run `php artisan serve` in one terminal window and leave it running throughout this practicum.

## C. Work Steps

### Step 1: A first route with a closure

Before using a controller, write the simplest possible route first: an address handled directly by a piece of code (a closure), with no separate class. This proves that a route is just a mapping from address → code, before controllers get added as an organizing layer.

Open `routes/web.php` and add the following lines (the full file contents after editing):

```php
<?php
// routes/web.php

use Illuminate\Support\Facades\Route;
```

```php
Route::get('/', function () {
    return view('welcome');
});

Route::get('/halo', function () {
    return 'Halo dari Simple POS';
});
```

Open `http://127.0.0.1:8000/halo` in your browser.

> ✅ **Checkpoint:** the page shows the plain text `Halo dari Simple POS`.

### Step 2: Observing HTTP in DevTools

Open your browser's DevTools (right-click → Inspect, or F12), switch to the **Network** tab, turn on **Preserve log**, then reload `http://127.0.0.1:8000/halo`.

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

| Address | Method | Status Code | Content-Type |
|---|---|---|---|
| `/halo` | | | |

Now open an address that's deliberately not registered, e.g. `http://127.0.0.1:8000/tidak-ada`, and record the new line in the same table.

| Address | Method | Status Code | Content-Type |
|---|---|---|---|
| `/tidak-ada` | | | |

> ⚠ **If it fails:** if the Network tab is empty after reloading, make sure **Preserve log** is on and the DevTools tab was already open **before** the page reloaded.

### Step 3: Observing HTTP with curl

DevTools shows HTTP through a visual interface; `curl` shows it as raw text, the same way you'll test APIs later.

```
curl -i http://127.0.0.1:8000/halo
```

> ✅ **Checkpoint:** the output's first line is `HTTP/1.1 200 OK`, followed by other headers (`Content-Type`, etc.), then a blank line, then the body `Halo dari Simple POS`.

### Step 4: Creating a TransactionController

A closure is fine for one simple route, but it's not practical once the logic grows. A controller groups related request-handling methods together in one class.

```
php artisan make:controller TransactionController
```

> ✅ **Checkpoint:** a new file appears at `app/Http/Controllers/TransactionController.php`, its contents an empty class like this:

```php
<?php

namespace App\Http\Controllers;

class TransactionController extends Controller
{
    //
}
```

### Step 5: Filling in stub methods

Simple POS will eventually have four methods on this controller: `create` (shows the cashier page), `store` (saves a new transaction), `index` (shows the transaction list), and `show` (shows one transaction's detail). Real views (Blade) aren't covered until Meeting 3, so for now each method just returns plain text as proof the flow is correct.

Replace the contents of `app/Http/Controllers/TransactionController.php` with:

```php
<?php

namespace App\Http\Controllers;

class TransactionController extends Controller
{
    public function create()
    {
        return 'Halaman kasir (belum ada tampilan)';
    }

    public function store()
    {
        return 'Transaksi disimpan (belum ada logika penyimpanan)';
    }

    public function index()
    {
        return 'Daftar transaksi';
    }

    public function show(string $id)
    {
        return "Detail transaksi #{$id}";
    }
}
```

### Step 6: Registering routes to the controller

Replace the contents of `routes/web.php` with (note the `use` line importing the controller):

```php
<?php
// routes/web.php

use App\Http\Controllers\TransactionController;
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return view('welcome');
});

Route::get('/halo', function () {
    return 'Halo dari Simple POS';
});

Route::get('/pos', [TransactionController::class, 'create'])
    ->name('pos.create');
Route::post('/pos', [TransactionController::class, 'store'])
    ->name('transactions.store');
Route::get('/transactions', [TransactionController::class, 'index'])
    ->name('transactions.index');
```

Open `http://127.0.0.1:8000/pos` and `http://127.0.0.1:8000/transactions` in your browser.

> ✓ **Checkpoint:** `/pos` shows the text `Halaman kasir (belum ada tampilan)`; `/transactions` shows the text `Daftar transaksi`.

> ⚠ **If it fails:** a `Target class [TransactionController] does not exist` error means the `use App\Http\Controllers\TransactionController;` line was forgotten or misspelled. Double-check the top of the file.

### Step 7: Checking the route list

Instead of opening `routes/web.php` and checking one by one that routes are registered, Artisan provides a command that prints a summary table.

```
php artisan route:list
```

> ✓ **Checkpoint:** the table shows `GET /pos` and `POST /pos` as **two separate rows**, even though the address is identical: they're distinguished by their HTTP method, one shows the cashier form, the other processes it.

### Step 8: Wrapping routes with the `auth` middleware

A route that should only be reachable by logged-in users needs to be wrapped in middleware. Change the Simple POS route section in `routes/web.php` to:

```php
<?php
// routes/web.php

use App\Http\Controllers\TransactionController;
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return view('welcome');
});

Route::get('/halo', function () {
    return 'Halo dari Simple POS';
});

Route::middleware('auth')->group(function () {
    Route::get('/pos', [TransactionController::class, 'create'])
        ->name('pos.create');
    Route::post('/pos', [TransactionController::class, 'store'])
        ->name('transactions.store');
    Route::get('/transactions', [TransactionController::class, 'index'])
        ->name('transactions.index');
});
```

Reload `http://127.0.0.1:8000/pos`.

> ✓ **Checkpoint (this isn't a bug, read it through):** the page shows a `Route [login] not defined.` error. This is actually **proof the middleware works**: before the request reaches `TransactionController`, the `auth` middleware checks whether the sender is logged in, finds they aren't, then tries to redirect to a login page, which won't exist until the authentication meeting later. Run `php artisan route:list` once more and notice the middleware column now shows `auth` on the `/pos` and `/transactions` rows.

> ⚠ **If it fails (in the real sense):** if the error that shows up is instead `Class "auth" does not exist` or similar, check how `Route::middleware('auth')` is written: the middleware name must be exactly the string `'auth'`, not a class name.

### Step 9: A controller processes before it responds

A controller doesn't just pass a request through: it can process data before sending a response. Change the `create()` method in `TransactionController` to prove it:

```php
public function create()
{
    $waktu = now()->format('H:i:s');
```

```
    return "Halaman kasir dibuka pukul {$waktu}";
}
```

Reload `/pos` twice, a few seconds apart.

> ✓ **Checkpoint:** the displayed time changes on every reload, proof the controller runs fresh PHP code on every incoming request, rather than showing the same static page.

#### Step 10: Independent challenge

Add one new route `GET /pos/riwayat` to `routes/web.php`, inside the same `auth` middleware group as the other `/pos` routes, pointing to a new method named `riwayat` on `TransactionController`. The method just needs to return plain text, e.g. `return "Riwayat kasir";`.

> ✓ **Checkpoint:** `php artisan route:list` shows your new route with the address `pos/riwayat`, method GET, and the `auth` middleware, without opening `/pos/riwayat` in the browser (it would show the same `Route [login]` error as Step 8, and that's expected).

#### Step 11: Commit `increment 2`

```
git add .
git commit -m "increment 2: route dan controller transaksi"
git log --oneline
```

> ✓ **Checkpoint:** `git log --oneline` shows two lines: `increment 2: ...` on top, `increment 1: ...` below it.

## D. Tasks and Deliverables

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

- The output of `php artisan route:list` after Step 10 (showing the `/pos/riwayat` route).
- A screenshot of the DevTools Network tab for `/halo` (status 200) and `/tidak-ada` (status 404).
- The Step 2 table.
- The output of `git log --oneline` showing the `increment 2` commit.
- **Independent task:** explain in your own words, in 3-5 sentences: (a) why `GET /pos` and `POST /pos` are considered two different routes even though their URL address is identical, and (b) what would happen if the route `DELETE /produk/{id}` forgot to be wrapped in middleware that checks for an admin role.

## E. Grading Rubric

| Component | Weight | Full Marks (100%) | Minimum Marks |
|---|---|---|---|
| Work steps completed | 40% | All of Steps 1-11 done and working | Most steps done |
| Checkpoints verified | 30% | `route:list`, DevTools screenshots, and tables complete and correct | Some checkpoints proven |
| Independent task | 20% | Both explanations are accurate and self-written | An answer exists but is incomplete |
| Commit hygiene | 10% | The commit message is exactly `increment 2: route dan controller transaksi` | A commit exists, but the message is messy |