

REOL

A lehenga store with UPI checkout.

Stack	Plain PHP 8 and MariaDB. No framework, no Composer, no build step – the target is shared hosting
Status	Complete as a piece of work; not yet launched
Size	Fifteen commits, six documents, and one smoke test that buys a lehenga

The premise

Most small Indian retailers do not take cards online. They take UPI, and they confirm a payment by looking at their phone. REOL is a store built around that fact rather than around a payment gateway: a customer places an order, pays by UPI to the store's ID, uploads a screenshot of the confirmation, and a person on the admin side verifies it against their own banking app before the order moves. It is slower than a card gateway, it involves a human on every order, and it is exactly how the business runs.

Designing for that changes more than the checkout page. An order has a state between “placed” and “paid” that can last hours; the admin's first screen is the verification queue rather than the order list; a proof can be rejected and re-uploaded; a customer needs to see, on the tracking page, that their payment is waiting on a person and not lost.

The shape

Plain PHP 8 and MariaDB, no framework, no package manager, no build step. The deployment target is shared hosting, where the honest constraint is *upload files*, *import a .sql*, *edit a .env*, and every dependency is one more thing that can go wrong on a host you do not control. It is the same discipline as this site, for the same reason.

The storefront: products, collections, a bag, checkout, UPI payment, order tracking. The admin, which the codebase calls the parlour: products, collections, orders, payment verification, customers, discount codes, customer messages and settings.

Three staff roles – owner, manager, staff – are fixed in one configuration file rather than in a database table, so that who-can-do-what is answerable from a diff. A seeder ships demo credentials for every role, and one console command creates the first real owner. The layering is a front controller, services, and repositories over PDO, with the rule that a controller never writes SQL and a repository never renders HTML. Every write is CSRF-protected and every query is a prepared statement; on a codebase with no framework, those two rules have to be habits, because nothing else enforces them.

The comments explain why

Several comments in the codebase record a bug that has already happened once. The guest-cart merge, when someone signs in with items already in their bag and items already in their account's bag from last week. The clock mismatch between PHP and the database on a host where the two are configured to different time zones, which made an order appear to be placed before its discount code was created. Each is written so that the next person does not delete the odd-looking line as noise, because deleting it is how the bug happens again.

A comment that explains *what* a line does is a line that should have been clearer. A comment that explains *why* is the only kind worth writing.

The test that matters is a crawler

There are unit tests, but the guard that matters is a smoke test that drives a real purchase end to end over real HTTP against a running development server. It signs in as a customer and as an admin; adds to the bag; places an order; uploads a payment proof; verifies the proof from the admin side; and then visits every route the store has and checks that each renders. What it proves is that no page errors and that the flows connect.

That is the right test for a store, because the failure mode of a store is not a wrong number; it is a checkout that quietly stopped working after an unrelated change, and nobody notices until a customer does. It runs

before every push.

Where it stands

The feature set it set out to have, it has. It has not yet been launched, and the launch checklist is written: the day it goes live, that list becomes the work.