Pizza Ordering System

A Beginner's Step-by-Step Tutorial

PHP - MySQL - Bootstrap 5 - Git & GitHub

with Live Data Updates - 2026 Edition

Build a complete web application from scratch

Customer registration and login with secure password hashing.
Admin login with role-based access control.
Two pizzas at different prices, ordered with quantity.
Customer dashboard and admin dashboard, both with live updates.
Order approval workflow handled by the admin.
Bootstrap 5 responsive UI - minimum custom CSS.
Version control with Git and a GitHub portfolio repo.
Full SDLC walk-through, folder structure, and deployment.

Beginner-Friendly - Step by Step - 2026 Edition

Table of Contents

Chapter 1 - Introduction and What You Will Build

Welcome! In this tutorial you will build a real, working **Pizza Ordering System** from scratch. By the end you will have a website where customers can register, log in, browse a menu of two pizza types, place an order, and see the order status update **live on their dashboard**. An admin can log in to a separate dashboard, see new orders as they come in, and approve them with one click.

Who this tutorial is for

- You know basic HTML and CSS but want to build a real backend application.
- You have heard of PHP and MySQL and want to use them properly.
- You want a clean, modern project you can put on GitHub and show to people.
- You want to understand **why** we do each step, not just copy code.

What the finished system does

- Two pizza types on the menu: **Margherita** and **Pepperoni**, each at a different price.
- Customers register an account, log in, and place orders.

- Each customer has a private dashboard showing their order history.
- Admin logs in through a separate page and sees a live list of pending orders.
- Admin clicks **Approve** and the customer's dashboard updates automatically with no page refresh.
- Bootstrap 5 design - clean, mobile friendly, very little custom CSS.

What you will learn along the way

- How professional developers plan a project using the **SDLC** (Software Development Life Cycle).
- How to organise your folders and name your files so others can read your code.
- How to design a simple but correct database with three tables.
- How to write secure PHP that uses **prepared statements** and **password hashing**.
- How to use **AJAX polling** to make data update live on the page.
- How to use **Git** and **GitHub** to save and share your code.

> **TIP**
>
> This tutorial is hands-on. Open your code editor while you read, type the code yourself (don't just copy), and run it after each chapter. You learn ten times faster by typing.

What the finished app looks like

Before we write a single line of code, here is a preview of every screen you will build. Keep these pictures in mind as you work through the chapters - each one is a goal you will reach.

1. Home page (everyone sees this)

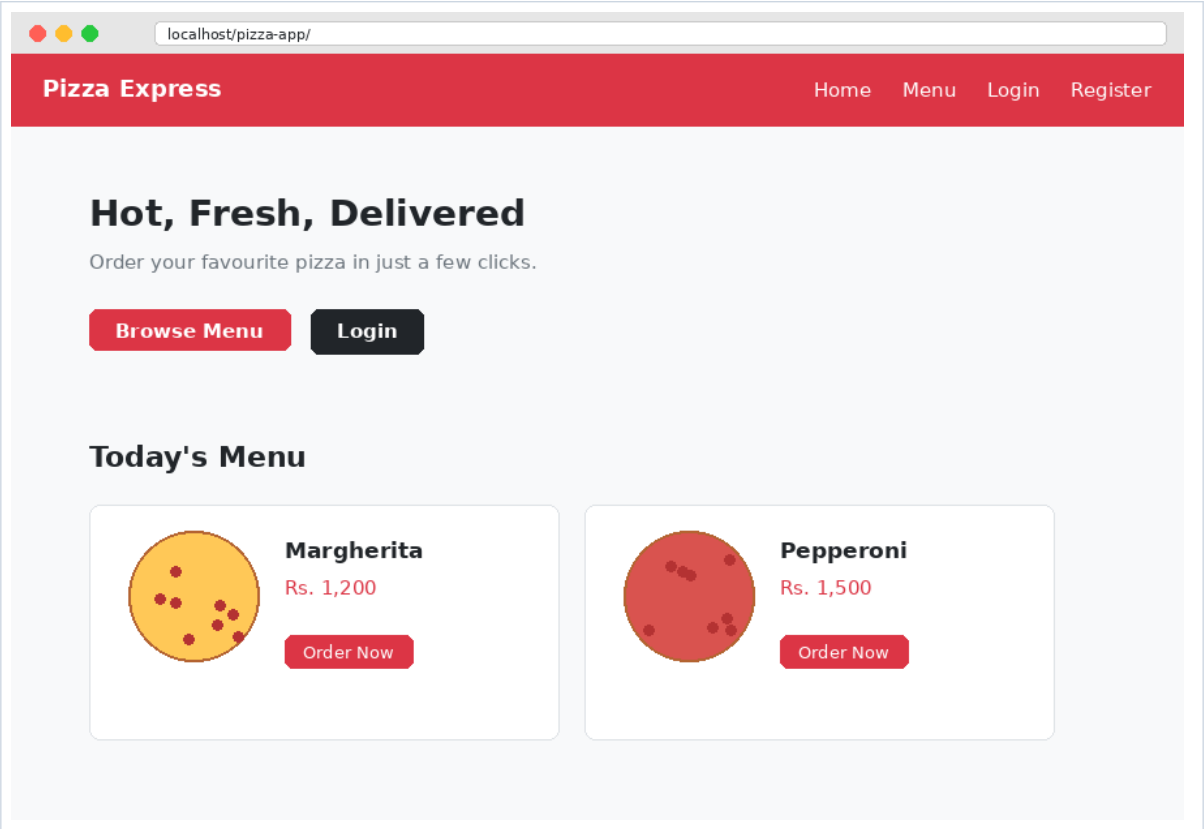


Figure 1 - Public home page with the menu preview and Login button.

2. Customer login

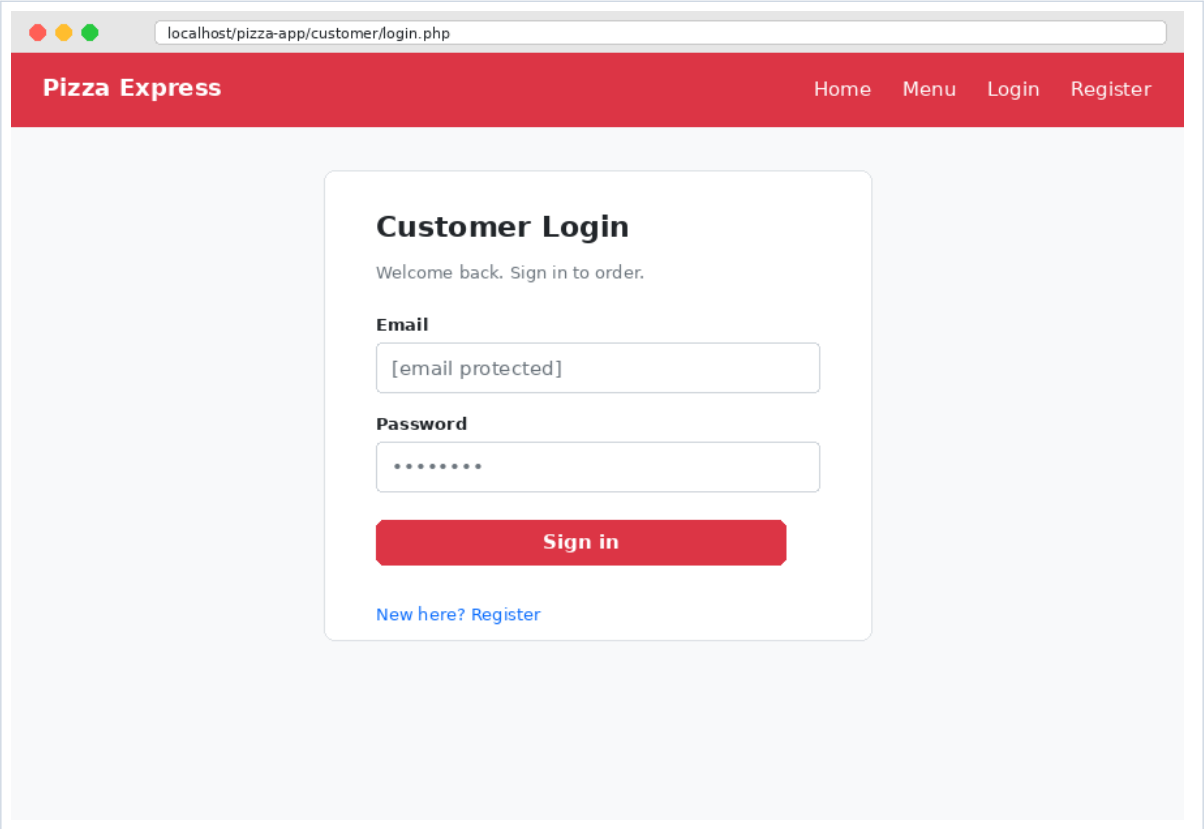


Figure 2 - Customer login form. Registration is a similar page.

3. Menu page (logged-in customer)

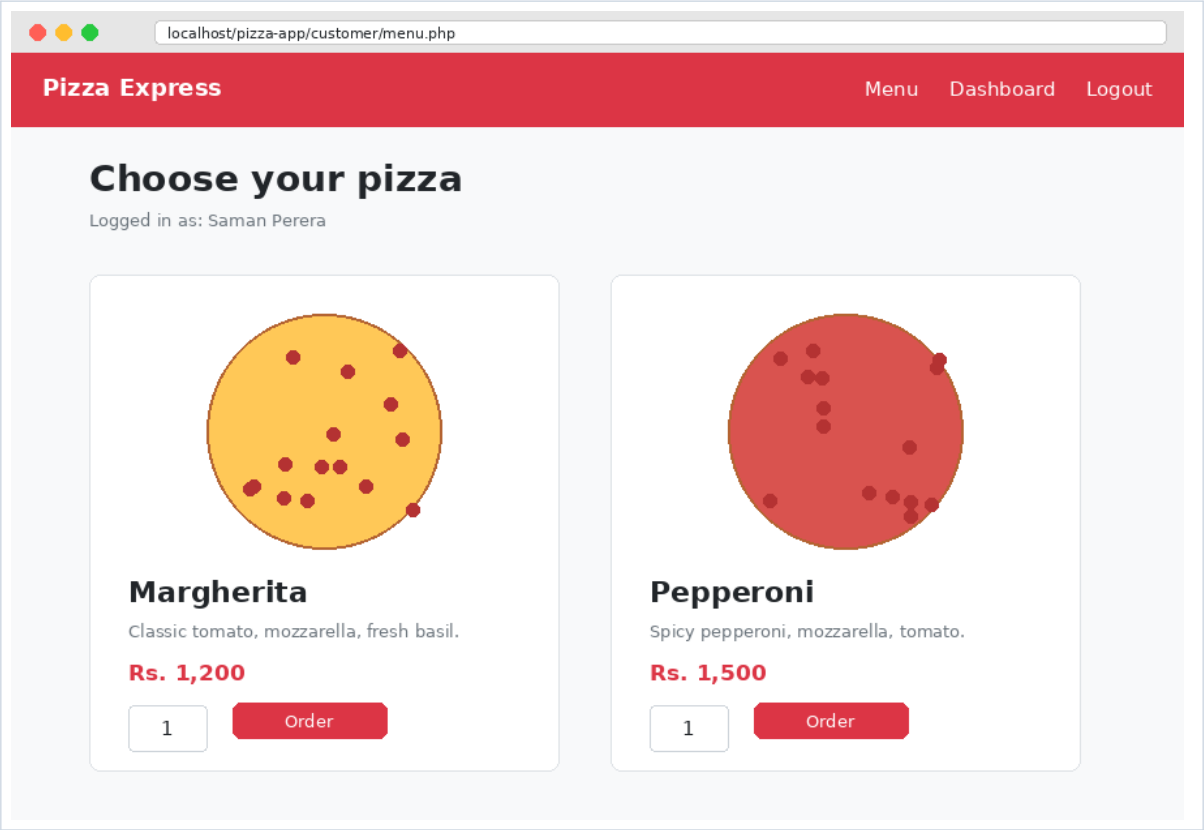


Figure 3 - Menu page. Customer picks a pizza and quantity, then orders.

4. Customer dashboard (live updates)

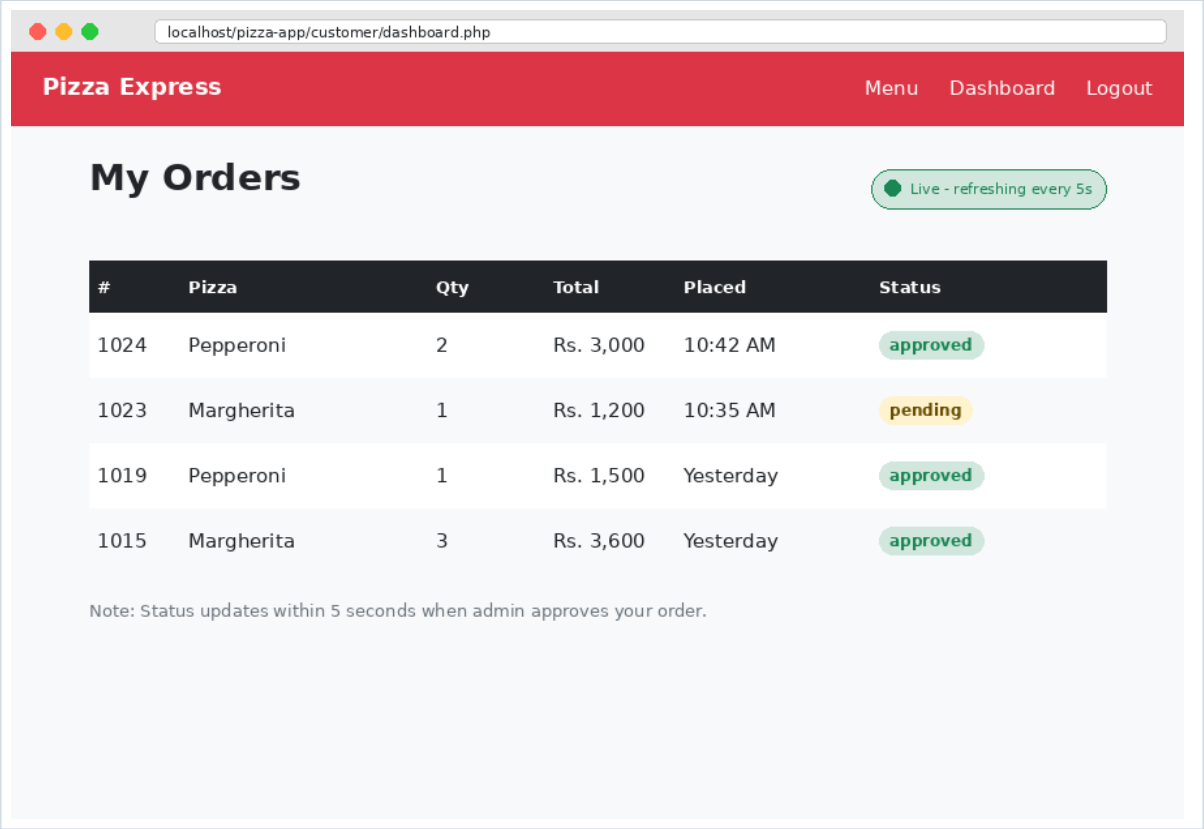


Figure 4 - Customer's order history. The status badge changes from yellow 'pending' to green 'approved' within 5 seconds, with no page reload.

5. Admin login (separate page)

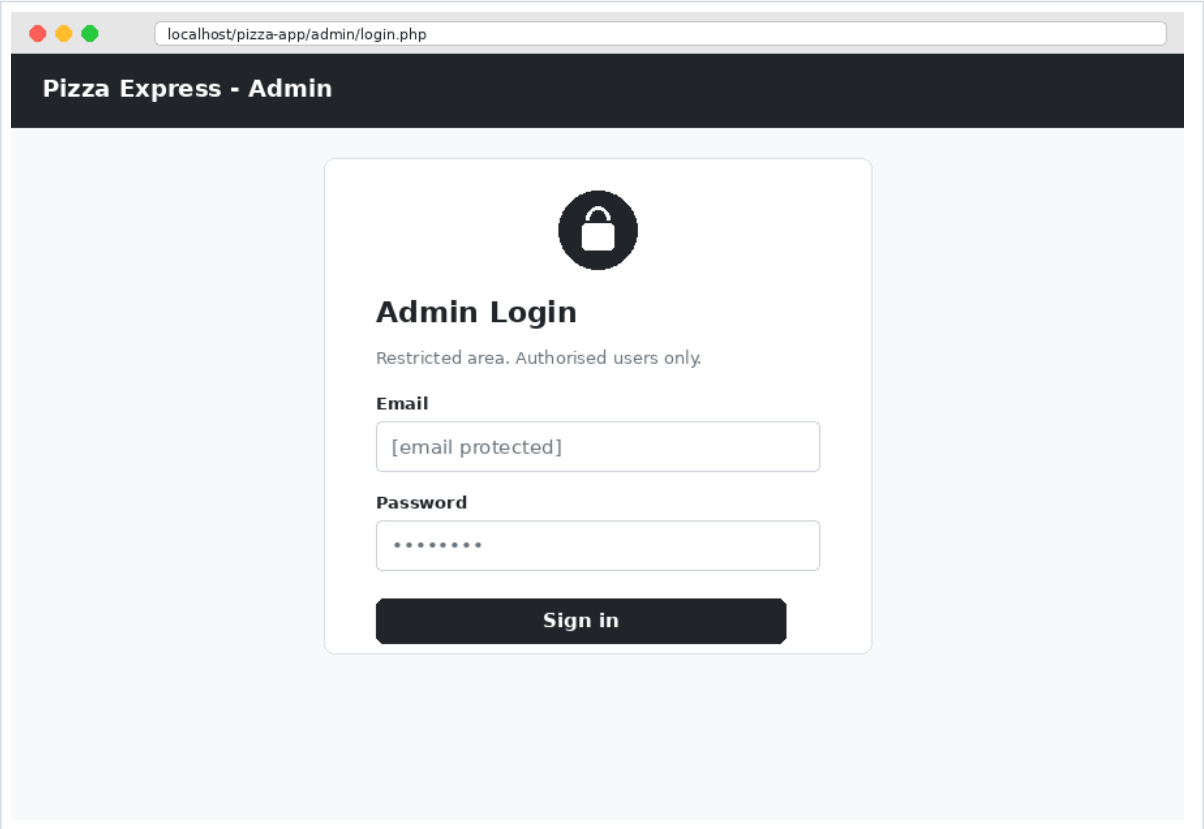


Figure 5 - Admin login page at /admin/login.php. Different URL, different theme - admin only.

6. Admin dashboard (sees every order)

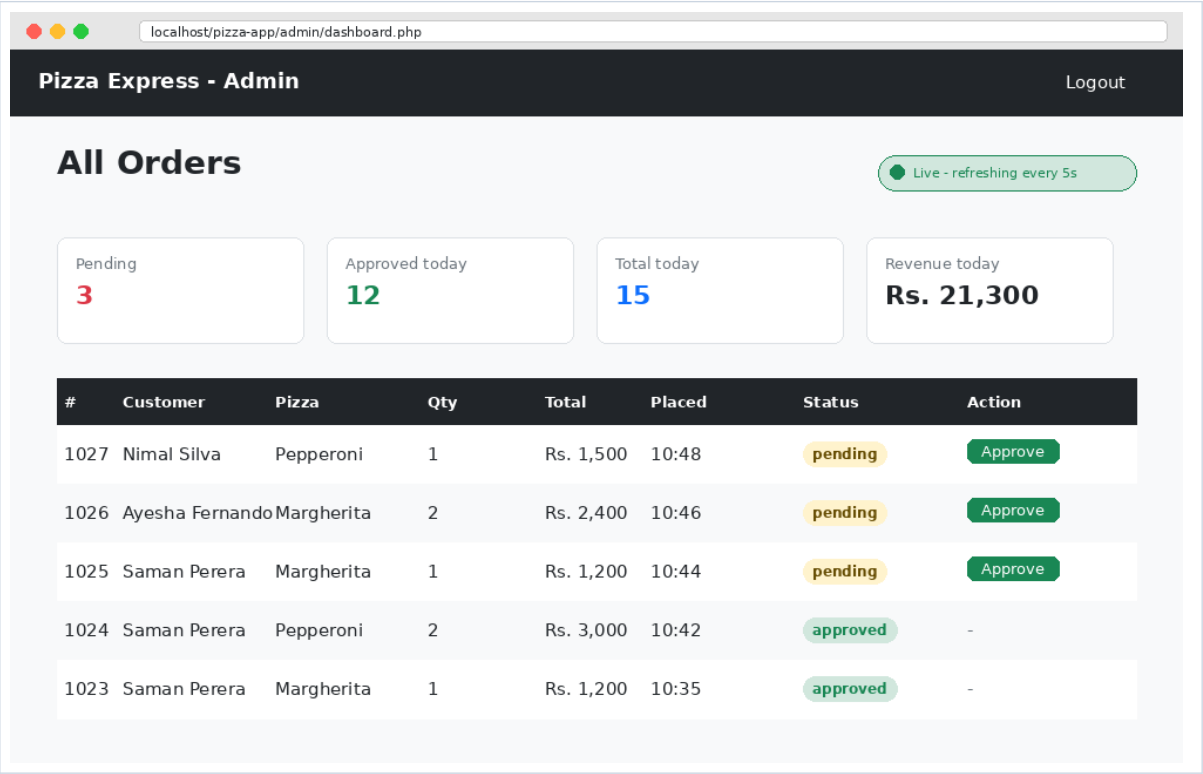


Figure 6 - Admin dashboard. Pending orders show an Approve button. The whole table refreshes every 5 seconds so new orders appear instantly.

How the live update flow works

Customer places an order on screen 3, sees it in their dashboard (screen 4) as **pending**. Admin sees it on screen 6 and clicks **Approve**. Within 5 seconds, the customer's dashboard automatically switches the badge to **approved**. No page reload, no email - just a smooth live update. We will build this exact flow in chapters 10-12.

Chapter 2 - Software Development Life Cycle (SDLC)

Before we write any code, let's talk about **how** professional software is built. The **SDLC** is a roadmap that takes you from "I have an idea" all the way to "my app is live and being used". It has six phases. Real teams follow these phases, and you will too - even on a small project.

The six phases

Phase	What happens	What we do in this project
1. Planning	Decide the goal, scope, and who the users are.	We list our two user types: customer and admin.
2. Requirements	Write down exactly what the system must do.	We list every feature: register, login, order, approve.
3. Design	Plan the database, folder layout, and screens.	We sketch tables: users, pizzas, orders. Plan folders.
4. Implementation	Write the code one feature at a time.	We build login first, then menu, then dashboard.
5. Testing	Try every flow and fix bugs before users see them.	We test as a customer, then as an admin.
6. Deployment & Maintenance	Put the app online and keep improving it.	We push to GitHub and discuss going live.

Why this matters for beginners

Beginners often jump straight to coding. The result is half-finished features, messy folders, and a database that has to be redesigned three times. By spending 30 minutes on phases 1 to 3 first, you save many hours later. Think of it like this: you would not start cooking a pizza without checking you have flour, cheese and tomato sauce. Same idea here.

The Agile twist (modern way)

Modern teams in 2026 don't do the six phases once and stop. They do them in small loops called **sprints**, usually 1 to 2 weeks each. In every sprint, you plan a small piece, design it, build it, and test it. We will follow that approach: each chapter is one mini-sprint that delivers a working feature.

> **NOTE**
>
> **SDLC = Plan, Requirements, Design, Build, Test, Deploy.** Memorise this. Every interviewer for a developer job will ask about it.

Chapter 3 - Tools You Need to Install

Install these on your computer before chapter 4. Take your time - getting tools set up correctly is half the battle. All of these are free.

Tool	Why we need it	Where to get it
XAMPP	Runs PHP and MySQL on your laptop so you can develop locally.	apachefriends.org
VS Code	Free code editor with great PHP support.	code.visualstudio.com
Git	Version control - saves snapshots of your code.	git-scm.com
GitHub account	Free online home for your code projects.	github.com
A modern browser	Chrome, Firefox or Edge - for testing your site.	Already on your computer.

Quick check after installation

Open a terminal (or Command Prompt on Windows) and run:

```
php --version
git --version
```

You should see version numbers for both. If you get a "command not found" error, the tool is not on your **PATH**. Re-open the terminal first; if it still fails, re-install and tick the box that says "Add to PATH" during setup.

Start XAMPP

Open the XAMPP Control Panel and click **Start** next to **Apache** and **MySQL**. Both indicators should turn green. Now open your browser and go to:

```
http://localhost
```

If you see the XAMPP welcome page, congratulations - you have a real web server running on your computer.

> **WARNING**
>
> On Windows, port 80 is sometimes used by Skype or IIS. If Apache won't start, open **Config -> Apache (httpd.conf)**, search for **Listen 80** and change it to **Listen 8080**. Then visit **http://localhost:8080** instead.

Chapter 4 - Project Planning and Requirements

This is phase 1 and 2 of the SDLC. We answer two questions: **who uses the system?** and **what must it do?** Skip this and you will rewrite everything later.

User roles

- **Customer** - registers, logs in, browses menu, places orders, sees own order status.
- **Admin** - logs in with admin account, sees all orders, approves pending ones.

Functional requirements

These are concrete things the system must do. Each one becomes a feature we will build.

1. A new visitor can create a customer account with name, email and password.
2. A registered customer can log in and log out.
3. A logged-in customer can see two pizzas with names, descriptions and prices.
4. A logged-in customer can place an order by choosing a pizza and quantity.
5. A logged-in customer can see all their past orders with status (Pending or Approved).
6. An admin can log in via a separate admin login page.
7. An admin can see every order from every customer in real time.
8. An admin can click Approve to change a Pending order to Approved.
9. When an order is approved, the customer's dashboard shows the new status without refreshing.

Non-functional requirements

These describe **how well** the system should behave, not what it does.

- **Security:** passwords stored hashed, all SQL uses prepared statements.
- **Usability:** works on a phone - Bootstrap 5 makes the layout responsive.
- **Performance:** dashboard polls for new data every 5 seconds. Light enough for cheap shared hosting.
- **Maintainability:** minimum custom CSS, clean folder structure, every file has one job.

Out of scope (we are NOT building these)

It's just as important to write down what you are *not* doing. It stops scope creep - the silent killer of beginner projects.

- Online payments (Stripe, PayPal etc.) - orders are cash on delivery for now.
- Email notifications.
- Pizza customisation (extra toppings) - only two fixed pizzas.
- Delivery tracking on a map.

> **TIP**

> Print or write this requirements list and stick it next to your monitor. Tick each item as you finish the matching chapter. This is exactly how professional Agile teams work.

Chapter 5 - Folder Structure and File Names

A clean folder structure is one of the biggest signs of a professional developer. Anyone - including future you, six months from now - should be able to open your project and instantly know where things live.

The structure we will use

Create this folder inside XAMPP's **htdocs** directory. On Windows that is usually **C:\xampp\htdocs\pizza-app**.

```
pizza-app/
|
+-- config/
|   `-- db.php                  ... database connection (used everywhere)
|
+-- includes/
|   +-- header.php              ... top of every page (navbar, Bootstrap CSS)
|   +-- footer.php              ... bottom of every page (Bootstrap JS)
|   `-- auth.php                ... helper: check if user is logged in
|
+-- customer/
|   +-- register.php            ... sign up form
|   +-- login.php               ... log in form
|   +-- logout.php              ... destroy session
|   +-- menu.php                ... pizza menu + order form
|   +-- place_order.php         ... receives the order POST
|   +-- dashboard.php           ... customer's order list (auto-refreshing)
|   `-- orders_api.php          ... returns JSON for live updates
|
+-- admin/
|   +-- login.php               ... admin login form
|   +-- logout.php              ... admin logout
|   +-- dashboard.php           ... all orders, approve buttons
|   +-- approve.php             ... receives the approve POST
|   `-- orders_api.php          ... JSON for the live admin view
|
+-- assets/
|   `-- style.css               ... only ~20 lines of custom CSS
|
+-- sql/
|   `-- schema.sql              ... create-database script (run once)
|
+-- index.php                   ... public landing page
+-- .gitignore                  ... tells Git what to ignore
`-- README.md                   ... project description for GitHub
```

Why this layout works

- **Separation of concerns:** customer code lives in *customer/*, admin code in *admin/*. Nothing leaks across.
- **config/** holds settings - never copy your DB password into other files.
- **includes/** holds reusable snippets - write once, include everywhere.
- **assets/** holds CSS, images, JS - anything the browser downloads directly.
- **sql/** holds the database script so anyone can recreate your DB.

File naming rules

- All lowercase: **menu.php** not **Menu.php**. Linux servers care about case.
- Use underscores or hyphens, never spaces: **place_order.php**.
- Name describes what it does: **approve.php** beats **page2.php** every time.
- Pages a user visits end in **.php**. Endpoints called by JavaScript end in **_api.php** by convention so you can spot them at a glance.

> **NOTE**
>
> Right now, just create the empty folders. We will fill in the files chapter by chapter. Having the skeleton in place first makes everything click into the right slot.

Chapter 6 - Database Design and SQL Setup

Every web app needs to remember things - users, products, orders. We use **MySQL**, the most popular open-source database. Our database has **three tables**. That's enough for this whole project.

The three tables

Table	Holds	Key columns
users	Customers and the admin account.	id, name, email, password, role
pizzas	The two pizzas on the menu.	id, name, description, price
orders	Every order placed.	id, user_id, pizza_id, quantity, status, created_at

How the tables connect

An order belongs to one user (the customer who placed it) and points to one pizza (the item ordered). We connect them with **foreign keys** - *orders.user_id* matches *users.id*, and *orders.pizza_id* matches *pizzas.id*. This is the core idea of a relational database.

The schema.sql file

Create a file at **sql/schema.sql** and paste the code below. We will run it once in phpMyAdmin to set everything up.

```
-- ============================================
-- Pizza Ordering System - Database Schema
-- ============================================

CREATE DATABASE IF NOT EXISTS pizza_app
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_unicode_ci;

USE pizza_app;

-- ----- Users (customers + admin) -----
CREATE TABLE users (
    id          INT AUTO_INCREMENT PRIMARY KEY,
    name        VARCHAR(100) NOT NULL,
    email       VARCHAR(150) NOT NULL UNIQUE,
    password    VARCHAR(255) NOT NULL,            -- stores a hash, not plain text
    role        ENUM('customer','admin') NOT NULL DEFAULT 'customer',
    created_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- ----- Pizzas (the menu) -----
CREATE TABLE pizzas (
    id          INT AUTO_INCREMENT PRIMARY KEY,
    name        VARCHAR(80) NOT NULL,
    description VARCHAR(255) NOT NULL,
    price       DECIMAL(8,2) NOT NULL
);

-- ----- Orders -----
CREATE TABLE orders (
    id          INT AUTO_INCREMENT PRIMARY KEY,
```

```
    user_id     INT NOT NULL,
    pizza_id    INT NOT NULL,
    quantity    INT NOT NULL DEFAULT 1,
    status      ENUM('Pending','Approved') NOT NULL DEFAULT 'Pending',
    created_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id)  REFERENCES users(id),
    FOREIGN KEY (pizza_id) REFERENCES pizzas(id)
);

-- ----- Seed data -----
INSERT INTO pizzas (name, description, price) VALUES
('Margherita', 'Classic tomato, mozzarella & fresh basil',  9.50),
('Pepperoni',  'Pepperoni, mozzarella & rich tomato sauce', 11.50);

-- Admin account (password is 'admin123' - we update the hash in chapter 11)
INSERT INTO users (name, email, password, role) VALUES
('Site Admin', 'admin@pizza.test',
 'PLACEHOLDER_HASH_REPLACED_IN_CH11',
 'admin');
```

Run the schema

1. Open **http://localhost/phpmyadmin** in your browser.
2. Click the **SQL** tab at the top.
3. Paste the entire contents of **schema.sql** into the box.
4. Click **Go**. You should see green success messages.
5. On the left side, click **pizza_app** - you should see three tables.

> **TIP**
>
> **utf8mb4** is the character set you should always use in 2026. It handles every emoji and every world language correctly. Older tutorials use plain **utf8** - that one is broken for emoji. Don't use it.

Quick database vocabulary

Term	What it means
Primary key	A column whose value is unique for each row (we use **id**).
Foreign key	A column pointing to another table's primary key.
Schema	The structure of your database - the tables and their columns.
Seed data	Initial rows you insert so the app has something to show.
ENUM	A column that can only be one of a fixed set of values.

Chapter 7 - Setting Up the Project

Now we wire up PHP to the database and create the shared header and footer that every page will include. Once these three files exist, every other chapter just builds on top.

config/db.php

This is the only file that knows your database password. Every other PHP file connects through this one. If you ever change your password, you change it here only.

```
<?php
// config/db.php  -  single database connection used everywhere

$DB_HOST = 'localhost';
$DB_NAME = 'pizza_app';
$DB_USER = 'root';     // default XAMPP user
$DB_PASS = '';         // default XAMPP password is empty

try {
    $pdo = new PDO(
        "mysql:host=$DB_HOST;dbname=$DB_NAME;charset=utf8mb4",
        $DB_USER,
        $DB_PASS,
        [
            PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES   => false,
        ]
    );
} catch (PDOException $e) {
    die('Database connection failed: ' . $e->getMessage());
}

// Start a session on every page that includes this file
if (session_status() === PHP_SESSION_NONE) {
    session_start();
}
```

Why PDO?

PDO (PHP Data Objects) is the modern way to talk to a database in PHP. It gives us **prepared statements** for free, which protect us from SQL injection - the #1 security mistake beginners make. Old tutorials use **mysqli** or even **mysql_***. In 2026, use PDO.

includes/header.php

```
<?php require_once __DIR__ . '/../config/db.php'; ?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Pizza Express</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
          rel="stylesheet">
    <link href="/pizza-app/assets/style.css" rel="stylesheet">
</head>
<body class="bg-light">

<nav class="navbar navbar-dark bg-danger">
  <div class="container">
```

```
    <a class="navbar-brand fw-bold" href="/pizza-app/index.php">Pizza Express</a>
    <div>
      <?php if (isset($_SESSION['user_id'])): ?>
        <span class="text-white me-3">
          Hi, <?= htmlspecialchars($_SESSION['user_name']) ?>
        </span>
        <a class="btn btn-light btn-sm"
           href="/pizza-app/<?= $_SESSION['role'] ?>/logout.php">Logout</a>
      <?php else: ?>
        <a class="btn btn-light btn-sm" href="/pizza-app/customer/login.php">Login</a>
      <?php endif; ?>
    </div>
  </div>
</nav>

<main class="container py-4">
```

includes/footer.php

```
</main>

<footer class="text-center text-muted py-4">
  <small>&copy; <?= date('Y') ?> Pizza Express  -  built for learning.</small>
</footer>

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js">
</script>
</body>
</html>
```

assets/style.css (the entire CSS file)

Yes - that's it. Bootstrap 5 already gives us almost everything we need. Custom CSS is for the few small touches Bootstrap doesn't cover.

```
.pizza-card {
    transition: transform 0.15s ease;
}
.pizza-card:hover {
    transform: translateY(-2px);
}
.status-pending  { color: #b45309; font-weight: 600; }
.status-approved { color: #15803d; font-weight: 600; }
```

index.php - landing page

```
<?php require_once 'includes/header.php'; ?>

<div class="text-center py-5">
    <h1 class="display-4 fw-bold">Pizza Express</h1>
    <p class="lead">Hot pizza, ordered online, approved by us.</p>
    <a href="customer/login.php"    class="btn btn-danger btn-lg me-2">Customer Login</a>
    <a href="customer/register.php" class="btn btn-outline-danger btn-lg">Sign Up</a>
</div>

<?php require_once 'includes/footer.php'; ?>
```

Test it

Open **http://localhost/pizza-app/** in your browser. You should see the red navbar, the welcome message and two buttons. The buttons don't work yet - we build those in the next chapter.

> **WARNING**
>
> If you see *Database connection failed*, MySQL is not running. Open XAMPP and click Start next to MySQL.

Chapter 8 - Customer Registration and Login

Authentication is the foundation of any user-aware app. We will use **password_hash()** to store passwords safely, and PHP's **$_SESSION** to remember who is logged in.

The flow

1. User fills the registration form, then POST to **register.php**.
2. We hash the password and INSERT into **users** table.
3. User fills the login form, then POST to **login.php**.
4. We look up the email, check the password with **password_verify()**.
5. If correct, we save *user_id*, *user_name*, *role* in **$_SESSION**.
6. Now every page can check **$_SESSION['user_id']** to know who is signed in.

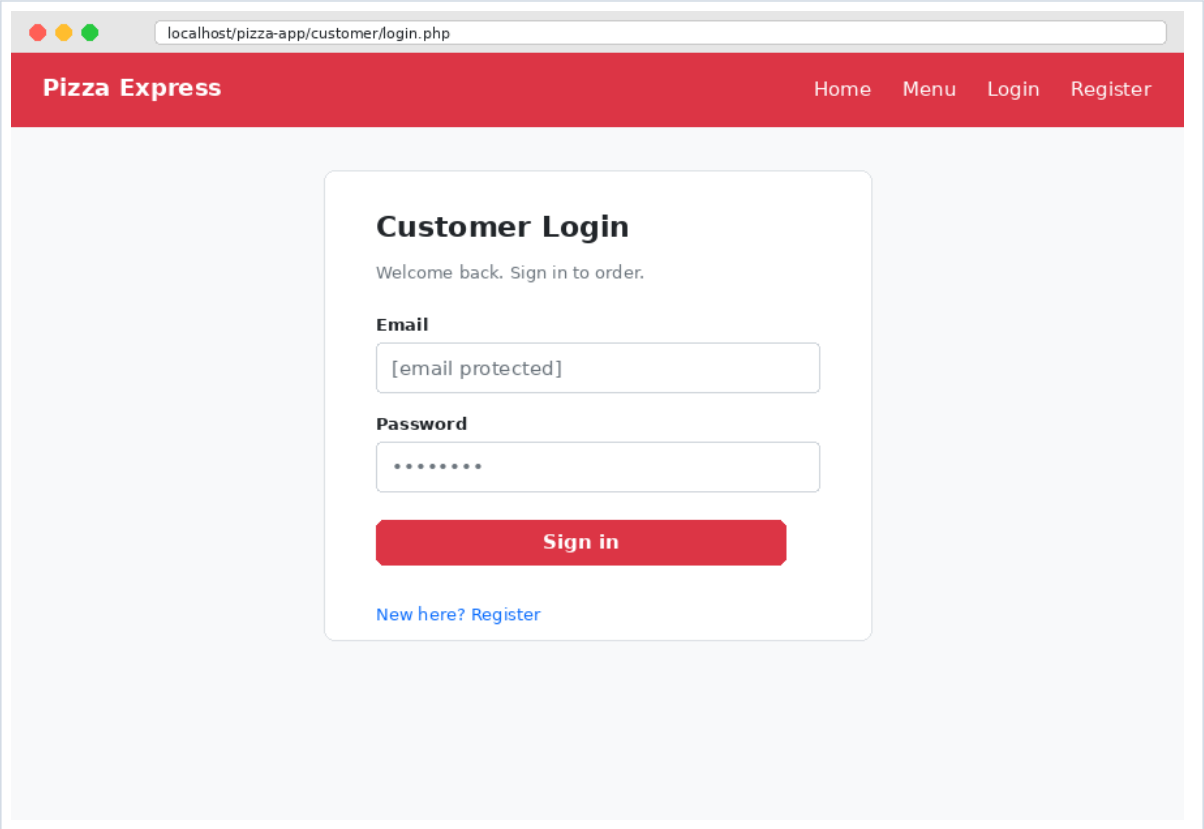
customer/register.php

```
<?php
require_once __DIR__ . '/../config/db.php';

$error = '';
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $name     = trim($_POST['name']     ?? '');
    $email    = trim($_POST['email']    ?? '');
    $password = $_POST['password'] ?? '';

    if ($name === '' || $email === '' || strlen($password) < 6) {
        $error = 'Fill all fields. Password must be at least 6 characters.';
    } else {
        // Check email is not taken
        $stmt = $pdo->prepare('SELECT id FROM users WHERE email = ?');
        $stmt->execute([$email]);
        if ($stmt->fetch()) {
            $error = 'That email is already registered.';
        } else {
            // Hash the password  -  NEVER store plain passwords
            $hash = password_hash($password, PASSWORD_DEFAULT);
            $stmt = $pdo->prepare(
                'INSERT INTO users (name, email, password, role)
                 VALUES (?, ?, ?, "customer")'
            );
            $stmt->execute([$name, $email, $hash]);
            header('Location: login.php?registered=1');
            exit;
        }
    }
}
require_once __DIR__ . '/../includes/header.php';
?>

<div class="row justify-content-center">
  <div class="col-md-5">
    <div class="card shadow-sm">
      <div class="card-body">
        <h3 class="card-title mb-3">Create your account</h3>

        <?php if ($error): ?>
          <div class="alert alert-danger"><?= htmlspecialchars($error) ?></div>
        <?php endif; ?>
```

```
        <form method="post">
          <div class="mb-3">
            <label class="form-label">Name</label>
            <input class="form-control" name="name" required>
          </div>
          <div class="mb-3">
            <label class="form-label">Email</label>
            <input class="form-control" type="email" name="email" required>
          </div>
          <div class="mb-3">
            <label class="form-label">Password (6+ chars)</label>
            <input class="form-control" type="password" name="password" required>
          </div>
          <button class="btn btn-danger w-100">Sign up</button>
          <p class="mt-3 mb-0 text-center">
            Already have an account? <a href="login.php">Log in</a>
          </p>
        </form>
      </div>
    </div>
  </div>
</div>

<?php require_once __DIR__ . '/../includes/footer.php'; ?>
```

customer/login.php

This is what we are building - the form on the right side. After we wrote the code, the page will look like Figure 2 in the preview. Type the code below into **customer/login.php**:



Customer login - what we are about to build.

```
<?php
require_once __DIR__ . '/../config/db.php';

$error = '';
```

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $email    = trim($_POST['email']    ?? '');
    $password = $_POST['password'] ?? '';

    $stmt = $pdo->prepare(
        'SELECT * FROM users WHERE email = ? AND role = "customer"'
    );
    $stmt->execute([$email]);
    $user = $stmt->fetch();

    if ($user && password_verify($password, $user['password'])) {
        $_SESSION['user_id']   = $user['id'];
        $_SESSION['user_name'] = $user['name'];
        $_SESSION['role']      = 'customer';
        header('Location: dashboard.php');
        exit;
    }
    $error = 'Wrong email or password.';
}
require_once __DIR__ . '/../includes/header.php';
?>

<div class="row justify-content-center">
  <div class="col-md-5">
    <div class="card shadow-sm">
      <div class="card-body">
        <h3 class="card-title mb-3">Customer Login</h3>

        <?php if (isset($_GET['registered'])): ?>
          <div class="alert alert-success">Account created  -  please log in.</div>
        <?php endif; ?>
        <?php if ($error): ?>
          <div class="alert alert-danger"><?= htmlspecialchars($error) ?></div>
        <?php endif; ?>

        <form method="post">
          <div class="mb-3">
            <label class="form-label">Email</label>
            <input class="form-control" type="email" name="email" required>
          </div>
          <div class="mb-3">
            <label class="form-label">Password</label>
            <input class="form-control" type="password" name="password" required>
          </div>
          <button class="btn btn-danger w-100">Log in</button>
          <p class="mt-3 mb-0 text-center">
            New here? <a href="register.php">Create an account</a>
          </p>
        </form>
      </div>
    </div>
  </div>
</div>

<?php require_once __DIR__ . '/../includes/footer.php'; ?>
```

customer/logout.php

```
<?php
require_once __DIR__ . '/../config/db.php';
session_destroy();
header('Location: /pizza-app/index.php');
exit;
```

includes/auth.php - the gatekeeper

Pages that need a logged-in user start with this single include. If the user isn't logged in, they get bounced back to the login page.

```
<?php
require_once __DIR__ . '/../config/db.php';

function require_login(string $required_role): void {
    if (
        !isset($_SESSION['user_id']) ||
        ($_SESSION['role'] ?? '') !== $required_role
    ) {
        $login_path = $required_role === 'admin'
            ? '/pizza-app/admin/login.php'
            : '/pizza-app/customer/login.php';
        header('Location: ' . $login_path);
        exit;
    }
}
```

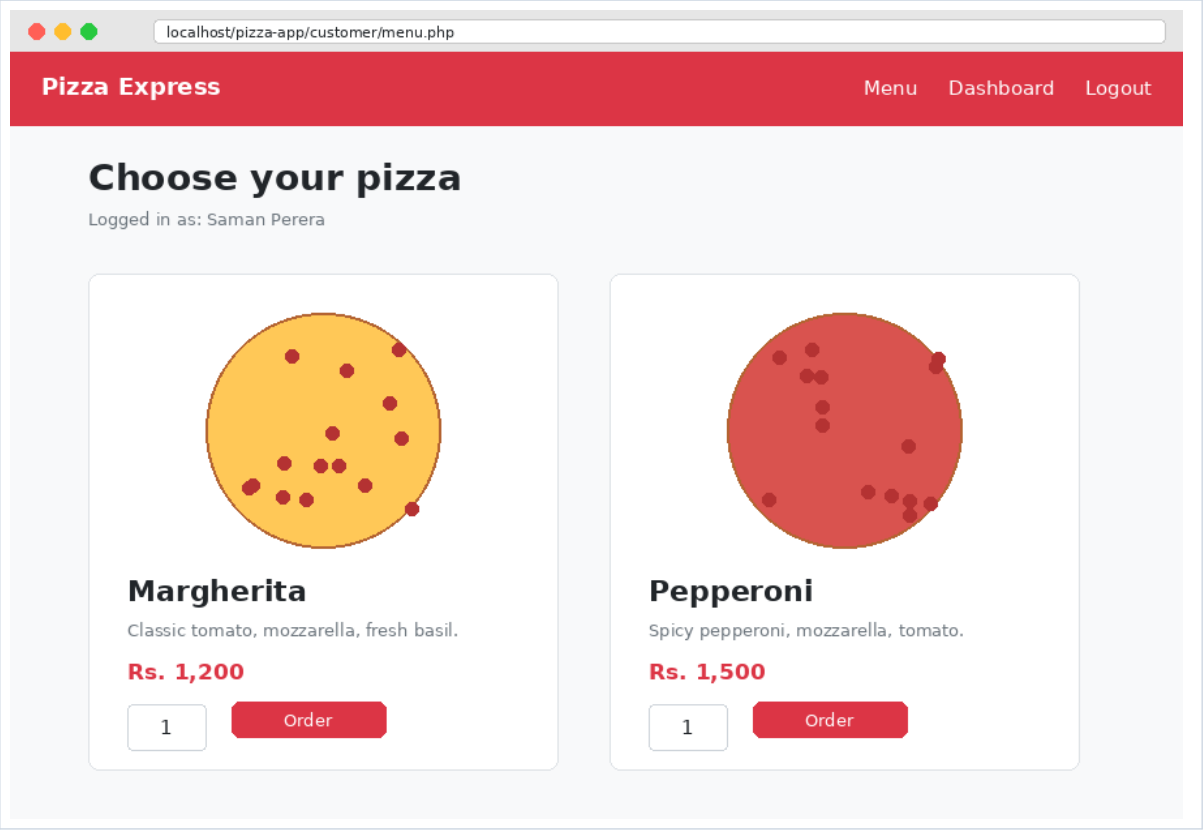
> **TIP**
>
> **Security checklist for this chapter:** (1) Passwords are hashed with **password_hash()**, never stored in plain text. (2) All SQL uses **prepared statements** - no string concatenation into queries. (3) All user-supplied output is wrapped in **htmlspecialchars()** to stop XSS attacks.

Chapter 9 - The Menu Page and Placing an Order

Now the fun part - the customer can see the two pizzas and order one. Notice how the menu is read from the database, not hard-coded. If you ever add a third pizza, you just INSERT a row - no PHP changes needed.

customer/menu.php

Here is the screen we are building - two pizza cards, each with its own quantity input and Order button:



The menu page that customer/menu.php will produce.

```
<?php
require_once __DIR__ . '/../includes/auth.php';
require_login('customer');

$pizzas = $pdo->query('SELECT * FROM pizzas ORDER BY id')->fetchAll();

require_once __DIR__ . '/../includes/header.php';
?>

<h2 class="mb-4">Our Menu</h2>

<div class="row g-4">
  <?php foreach ($pizzas as $pizza): ?>
    <div class="col-md-6">
      <div class="card pizza-card shadow-sm h-100">
        <div class="card-body">
          <div class="d-flex justify-content-between">
            <h4 class="card-title"><?= htmlspecialchars($pizza['name']) ?></h4>
            <span class="badge bg-danger fs-6">
              $<?= number_format($pizza['price'], 2) ?>
```

```
            </span>
          </div>
          <p class="text-muted">
            <?= htmlspecialchars($pizza['description']) ?>
          </p>

          <form method="post" action="place_order.php" class="d-flex gap-2">
            <input type="hidden" name="pizza_id" value="<?= $pizza['id'] ?>">
            <input class="form-control" type="number" name="quantity"
                   value="1" min="1" max="10" style="max-width: 90px;">
            <button class="btn btn-danger flex-grow-1">Order</button>
          </form>
        </div>
      </div>
    </div>
  <?php endforeach; ?>
</div>

<div class="mt-4">
  <a class="btn btn-outline-secondary" href="dashboard.php">My Orders</a>
</div>

<?php require_once __DIR__ . '/../includes/footer.php'; ?>
```

customer/place_order.php

This file *only* handles the POST. It writes to the database and then redirects. There is no HTML in it. Keeping POST handlers separate from view files is a clean habit that pays off in bigger projects.

```
<?php
require_once __DIR__ . '/../includes/auth.php';
require_login('customer');

if ($_SERVER['REQUEST_METHOD'] !== 'POST') {
    header('Location: menu.php');
    exit;
}

$pizza_id = (int)($_POST['pizza_id'] ?? 0);
$quantity = (int)($_POST['quantity'] ?? 0);

if ($pizza_id < 1 || $quantity < 1 || $quantity > 10) {
    header('Location: menu.php?error=invalid');
    exit;
}

$stmt = $pdo->prepare(
    'INSERT INTO orders (user_id, pizza_id, quantity)
     VALUES (?, ?, ?)'
);
$stmt->execute([$_SESSION['user_id'], $pizza_id, $quantity]);

header('Location: dashboard.php?placed=1');
exit;
```

Walk through what happens

1. Customer clicks **Order** on the menu page.
2. Browser POSTs **pizza_id** and **quantity** to **place_order.php**.
3. PHP checks the user is logged in as a customer.
4. PHP validates the inputs are sensible numbers.

5. PHP runs an INSERT into the **orders** table with status defaulting to **Pending**.

6. PHP redirects to the customer dashboard with a success flag.

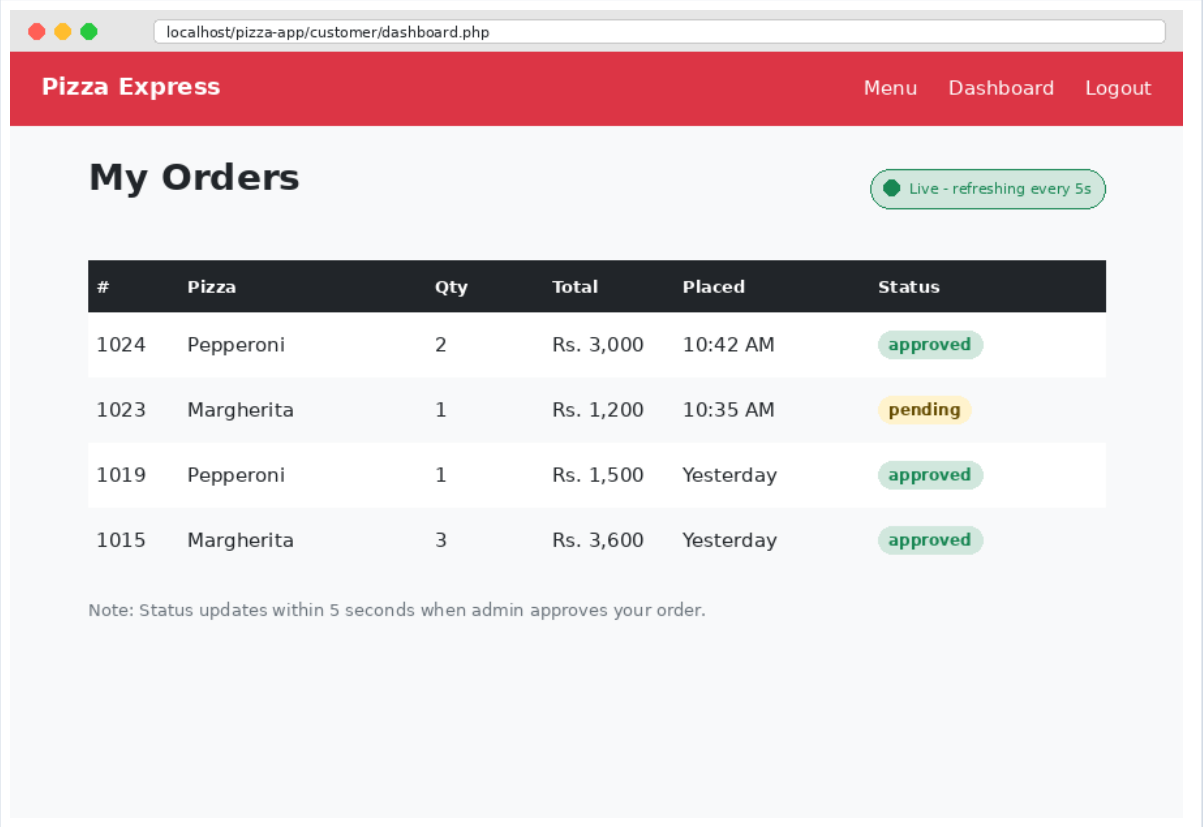
> **TIP**
>
> Notice **(int)$_POST['pizza_id']**. Casting to int means even if someone tries to send *'; DROP TABLE pizzas; --* as a value, it becomes the number 0 and the order silently fails. Defence in depth - even though prepared statements already protect us, we still validate types.

Chapter 10 - Customer Dashboard with Live Updates

This is the page the customer stares at after ordering. They want to see the moment the admin approves their pizza. We make this happen with **AJAX polling** - the page asks the server every 5 seconds "any changes?" and updates the screen if there are.

customer/dashboard.php

Goal: a clean orders table that updates itself. Watch the green *Live* badge in the top right - that is our way of telling the customer the page is refreshing automatically.



Customer dashboard with live status updates.

```
<?php
require_once __DIR__ . '/../includes/auth.php';
require_login('customer');
require_once __DIR__ . '/../includes/header.php';
?>

<h2 class="mb-3">My Orders</h2>

<?php if (isset($_GET['placed'])): ?>
  <div class="alert alert-success">
    Order placed! It will turn green once the kitchen approves it.
  </div>
<?php endif; ?>

<a href="menu.php" class="btn btn-danger mb-3">+ Order another pizza</a>

<div class="card shadow-sm">
  <div class="card-body p-0">
```

```
    <table class="table mb-0" id="ordersTable">
      <thead class="table-light">
        <tr>
          <th>#</th><th>Pizza</th><th>Qty</th>
          <th>Total</th><th>Status</th><th>When</th>
        </tr>
      </thead>
      <tbody>
        <!-- rows are filled in by JavaScript -->
      </tbody>
    </table>
  </div>
</div>

<p class="text-muted small mt-2">
  <span id="liveDot" class="text-success">[live]</span>
  Auto-refreshing every 5 seconds.
</p>

<script>
async function loadOrders() {
    try {
        const res  = await fetch('orders_api.php');
        const data = await res.json();

        const tbody = document.querySelector('#ordersTable tbody');
        if (data.length === 0) {
            tbody.innerHTML =
              '<tr><td colspan="6" class="text-center text-muted py-4">' +
              'No orders yet. Pick a pizza from the menu!</td></tr>';
            return;
        }

        tbody.innerHTML = data.map(o => `
            <tr>
                <td>${o.id}</td>
                <td>${o.pizza_name}</td>
                <td>${o.quantity}</td>
                <td>$${(o.price * o.quantity).toFixed(2)}</td>
                <td><span class="status-${o.status.toLowerCase()}">
                    ${o.status}
                </span></td>
                <td><small>${o.created_at}</small></td>
            </tr>
        `).join('');
    } catch (e) {
        document.getElementById('liveDot').className = 'text-danger';
    }
}

loadOrders();
setInterval(loadOrders, 5000);   // <-- the live update
</script>

<?php require_once __DIR__ . '/../includes/footer.php'; ?>
```

customer/orders_api.php - JSON for the live table

This file does NOT include the header or footer. It returns pure JSON. JavaScript on the dashboard page calls it every 5 seconds.

```
<?php
require_once __DIR__ . '/../includes/auth.php';
require_login('customer');
```

```
header('Content-Type: application/json');

$stmt = $pdo->prepare("
    SELECT o.id, o.quantity, o.status,
           DATE_FORMAT(o.created_at, '%d %b %H:%i') AS created_at,
           p.name  AS pizza_name,
           p.price AS price
      FROM orders o
      JOIN pizzas p ON o.pizza_id = p.id
     WHERE o.user_id = ?
     ORDER BY o.id DESC
");
$stmt->execute([$_SESSION['user_id']]);

echo json_encode($stmt->fetchAll());
```

Why polling instead of WebSockets?

WebSockets give true push, but they need extra server software (Node, Ratchet, or a separate service). Polling works on plain shared hosting with zero extra setup. For a pizza shop with maybe 50 orders an hour, polling every 5 seconds is more than fast enough - and the code is 30 lines instead of 300.

> **TIP**
>
> Open the dashboard in two browser tabs and place an order in one. After 5 seconds, the second tab shows the new row by itself. That is your live update working - and you wrote it in about 20 lines of JavaScript.

Chapter 11 - Admin Login and Dashboard

The admin is a special user with **role = 'admin'** in the users table. There is no admin signup page (we do not want random people becoming admin). Instead, we insert the admin account directly into the database with a hashed password.

Step 1 - Generate a hashed password

Create a temporary file **make_hash.php** in the project root. We will delete it after using it once.

```
<?php
// make_hash.php - run once, then DELETE this file
$password = 'admin123';  // change to whatever you want
echo password_hash($password, PASSWORD_DEFAULT);
```

Open **http://localhost/pizza-app/make_hash.php** in your browser. You will see a long string starting with *$2y$*. Copy it.

> **WARNING**
>
> WARNING: Delete make_hash.php as soon as you have copied the hash. Leaving it on a live server is a security hole.

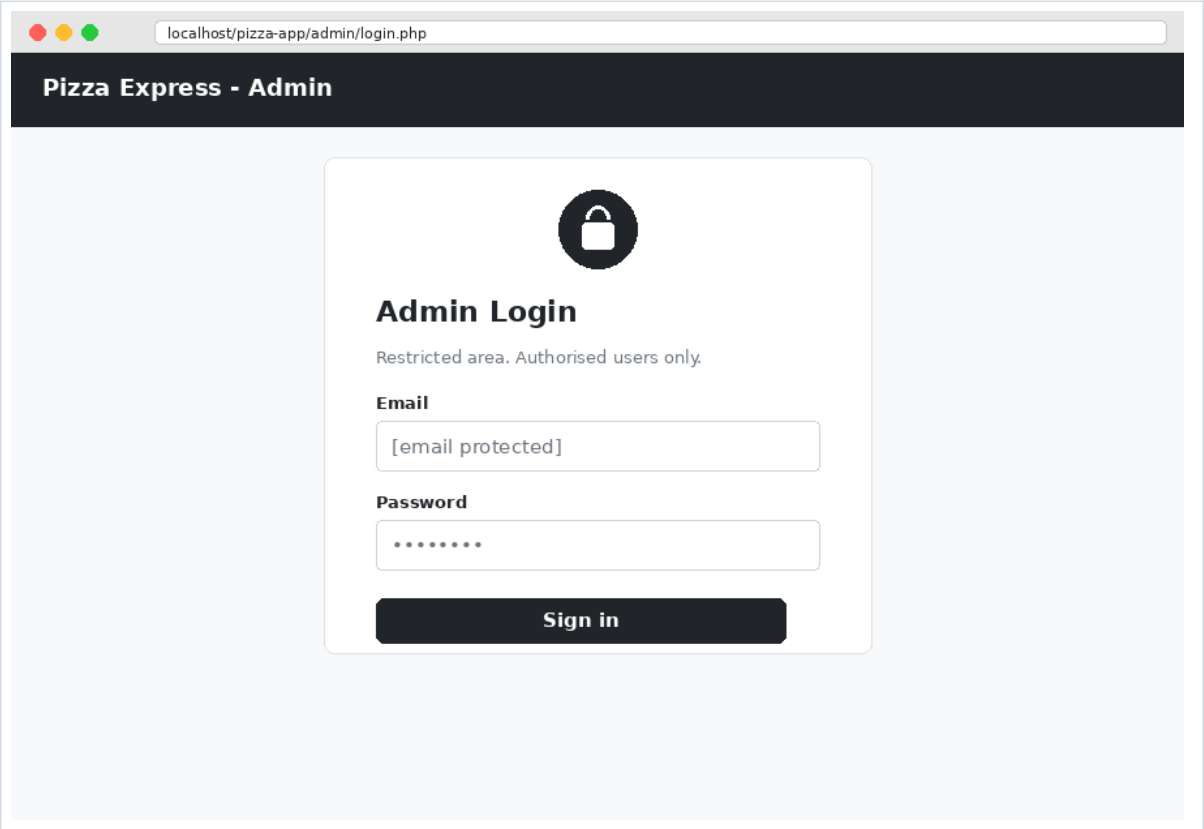
Step 2 - Insert the admin row in phpMyAdmin

Open phpMyAdmin, choose the **pizza_app** database, click the **SQL** tab, and run this query (paste your real hash where shown):

```
INSERT INTO users (name, email, password, role) VALUES (
    'Site Admin',
    '[email protected]',
    'PASTE_THE_HASH_HERE',
    'admin'
);
```

Step 3 - admin/login.php

The admin login is almost the same as the customer login, but it checks that the role is 'admin' before letting the person in. Notice the dark navbar - a small visual cue that this is the admin area:



Admin login - separate URL, separate look.

```
<?php
// admin/login.php
require_once __DIR__ . '/../config/db.php';
require_once __DIR__ . '/../includes/header.php';

$error = '';

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $email = trim($_POST['email'] ?? '');
    $password = $_POST['password'] ?? '';

    $stmt = $pdo->prepare("SELECT * FROM users WHERE email = ?");
    $stmt->execute([$email]);
    $user = $stmt->fetch();

    if ($user && $user['role'] === 'admin'
        && password_verify($password, $user['password'])) {
        $_SESSION['user_id']   = $user['id'];
        $_SESSION['user_name'] = $user['name'];
        $_SESSION['user_role'] = 'admin';
        header('Location: dashboard.php');
        exit;
    } else {
        $error = 'Invalid admin credentials.';
    }
}
?>

<div class="row justify-content-center">
  <div class="col-md-5">
    <h2 class="mb-4">Admin Login</h2>
    <?php if ($error): ?>
      <div class="alert alert-danger"><?= htmlspecialchars($error) ?></div>
    <?php endif; ?>
```

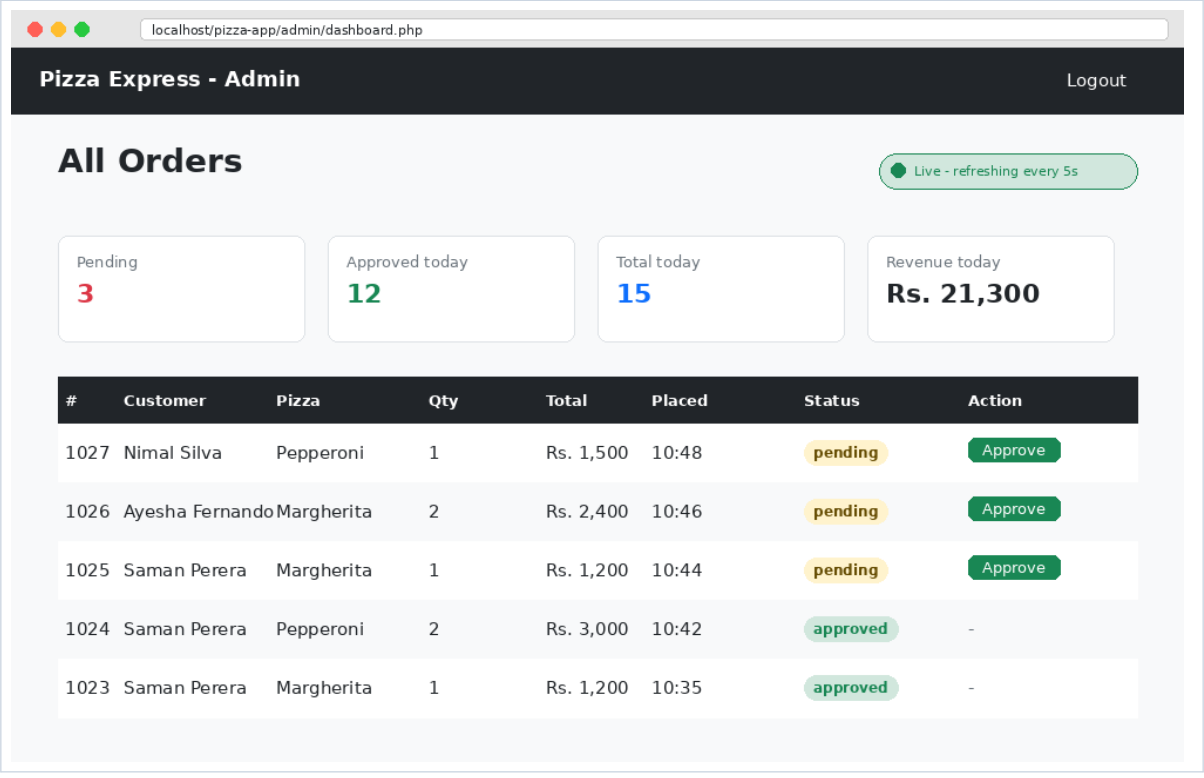
```
    <form method="post">
      <div class="mb-3">
        <label class="form-label">Email</label>
        <input type="email" name="email" class="form-control" required>
      </div>
      <div class="mb-3">
        <label class="form-label">Password</label>
        <input type="password" name="password" class="form-control" required>
      </div>
      <button type="submit" class="btn btn-dark w-100">Sign in</button>
    </form>
  </div>
</div>

<?php require_once __DIR__ . '/../includes/footer.php'; ?>
```

Step 4 - admin/logout.php

```
<?php
// admin/logout.php
session_start();
session_destroy();
header('Location: login.php');
exit;
```

Step 5 - admin/dashboard.php

The admin dashboard shows **every order from every customer**. Pending orders have an Approve button. Like the customer dashboard, the table refreshes itself every 5 seconds so new orders appear without a reload.



Admin dashboard - the command centre of the app.

```
<?php
// admin/dashboard.php
require_once __DIR__ . '/../includes/auth.php';
require_admin();  // see note below
```

```
require_once __DIR__ . '/../includes/header.php';
?>

<h2 class="mb-4">Admin Dashboard - All Orders</h2>

<div class="table-responsive">
  <table class="table table-striped align-middle">
    <thead class="table-dark">
      <tr>
        <th>#</th><th>Customer</th><th>Pizza</th><th>Qty</th>
        <th>Total</th><th>Placed</th><th>Status</th><th>Action</th>
      </tr>
    </thead>
    <tbody id="ordersBody">
      <tr><td colspan="8" class="text-center">Loading...</td></tr>
    </tbody>
  </table>
</div>

<script>
async function loadOrders() {
  const res = await fetch('orders_api.php');
  const data = await res.json();
  const body = document.getElementById('ordersBody');
  if (!data.length) {
    body.innerHTML = '<tr><td colspan="8" class="text-center">No orders yet.</td></tr>';
    return;
  }
  body.innerHTML = data.map(o => `
    <tr>
      <td>${o.id}</td>
      <td>${o.customer}</td>
      <td>${o.pizza}</td>
      <td>${o.quantity}</td>
      <td>Rs. ${o.total}</td>
      <td>${o.created_at}</td>
      <td><span class="badge bg-${o.status === 'approved' ? 'success' : 'warning text-dark'}">${o.stat
      <td>
        ${o.status === 'pending'
          ? `<button class="btn btn-sm btn-success" onclick="approve(${o.id})">Approve</button>`
          : '-'}
      </td>
    </tr>
  `).join('');
}

async function approve(id) {
  const res = await fetch('approve.php?id=' + id);
  if (res.ok) loadOrders();
}

loadOrders();
setInterval(loadOrders, 5000);
</script>

<?php require_once __DIR__ . '/../includes/footer.php'; ?>
```

Adding require_admin() to auth.php

Open **includes/auth.php** and add this function at the bottom:

```
function require_admin() {
    require_login();
    if (($_SESSION['user_role'] ?? '') !== 'admin') {
        header('Location: /pizza-app/index.php');
```

```
        exit;
    }
}
```

admin/orders_api.php - the data feed

```
<?php
// admin/orders_api.php
require_once __DIR__ . '/../includes/auth.php';
require_admin();
require_once __DIR__ . '/../config/db.php';

header('Content-Type: application/json');

$stmt = $pdo->query("
    SELECT o.id, o.quantity, o.status, o.created_at,
           u.name AS customer,
           p.name AS pizza, p.price,
           (p.price * o.quantity) AS total
    FROM orders o
    JOIN users u  ON u.id  = o.user_id
    JOIN pizzas p ON p.id = o.pizza_id
    ORDER BY o.created_at DESC
");
echo json_encode($stmt->fetchAll());
```

Chapter 12 - The Approve Workflow

When the admin clicks **Approve**, the JavaScript fires a request to **approve.php?id=...**. That script flips the status from 'pending' to 'approved'. Next time the customer dashboard polls (within 5 seconds), the green badge appears on their screen.

admin/approve.php

```
<?php
// admin/approve.php
require_once __DIR__ . '/../includes/auth.php';
require_admin();
require_once __DIR__ . '/../config/db.php';

$id = (int)($_GET['id'] ?? 0);
if ($id <= 0) {
    http_response_code(400);
    exit('Bad request');
}

$stmt = $pdo->prepare(
    "UPDATE orders SET status = 'approved' WHERE id = ? AND status = 'pending'"
);
$stmt->execute([$id]);

echo 'ok';
```

> **NOTE**
>
> NOTE: The query has a safety check - **WHERE id = ? AND status = 'pending'**. If the order was already approved (or does not exist), nothing happens. This prevents double-approvals and weird race conditions.

End-to-end test

Time to see the whole loop in action. Open two browsers side by side:

1. Left browser: log in as a customer (the account you registered earlier).
2. Right browser (use Incognito or a different browser): log in as admin at **/pizza-app/admin/login.php**.
3. Customer side: go to the menu, place an order for any pizza.
4. Watch the admin dashboard - within 5 seconds the new pending order appears.
5. Click **Approve** on the admin side.
6. Watch the customer dashboard - within 5 seconds the badge turns green.

> **TIP**
>
> TIP: If something does not update, open the browser DevTools (F12) -> Network tab. Look for orders_api.php requests every 5 seconds. If they are red, click one to see the error message.

What you have built so far

At this point your app has:

- User registration with hashed passwords.
- Customer login and logout.
- Admin login (separate role-checked entry).
- A menu page with two pizzas at different prices.
- Order placement with quantity selection.
- Customer dashboard with live status updates.
- Admin dashboard with live order feed.
- One-click order approval.

That is a complete **CRUD** application - Create (orders), Read (menus, dashboards), Update (approval), and the foundations are laid for Delete. Pat yourself on the back.

Chapter 13 - How Live Updates Work (Deep Dive)

The dashboards feel *live* but there is no magic - they just keep asking the server "any news?" every 5 seconds. This pattern is called **polling**. It is the simplest live-data technique and the right choice for a beginner project.

The polling pattern in plain English

1. Browser asks the server for the latest data (an HTTP request).
2. Server runs a query and sends back JSON.
3. Browser updates the table.
4. Wait 5 seconds.
5. Repeat from step 1.

The two lines that make it happen are at the bottom of each dashboard:

```
loadOrders();              // run once on page load
setInterval(loadOrders, 5000); // run again every 5 seconds
```

Polling vs other techniques

Technique	How it works	When to use
Polling	Browser asks every N seconds.	Beginner projects, low traffic, simple stack.
Long polling	Server holds the request until data changes.	Slightly faster updates, more server load.
SSE (Server-Sent Events)	Server pushes a stream to the browser.	One-way live feeds, news tickers.
WebSockets	Two-way persistent connection.	Chat apps, multiplayer games, real-time trading.

> **NOTE**
>
> NOTE: 5 seconds is a fine default. For a busy pizza shop you could drop it to 2 seconds. Going below 1 second is rarely worth the server load - switch to WebSockets at that point.

Why we did not use WebSockets

WebSockets would update instantly, but they require:

- A long-running PHP process (or Node.js, or Ratchet library).
- Extra server configuration.
- More complex client and server code.

For a tutorial project that runs on plain XAMPP and shared hosting, polling is the right call. You can always upgrade later.

Making polling smarter (optional)

One small improvement: only re-render the table if the data actually changed. Compare the new JSON to the previous one and skip the DOM update if they match. For two pizzas and a handful of orders, it does not matter - but for bigger lists it keeps the page from flickering.

Chapter 14 - Git and GitHub Step by Step

Git is a version control system - it remembers every change you make. **GitHub** is a website that hosts Git projects online. Together they give you backup, history, and a portfolio link you can show employers.

The five commands you will use 95% of the time

Command	What it does
git init	Start tracking a folder with Git.
git add .	Stage all changed files for the next commit.
git commit -m "msg"	Save a snapshot with a message.
git push	Upload commits to GitHub.
git pull	Download the latest commits from GitHub.

Step 1 - Create a .gitignore file

Some files should **never** be uploaded - your database password, log files, IDE settings. Create a file called **.gitignore** in the project root with these lines:

```
.vscode/
.idea/
*.log
node_modules/
config/secrets.php
```

> **WARNING**
>
> WARNING: If your db.php contains a real password, either rename it to secrets.php and gitignore that, or use environment variables. Never commit production passwords to GitHub.

Step 2 - Initialize the repo

Open a terminal in your project folder and run:

```
cd C:\xampp\htdocs\pizza-app
git init
git add .
git commit -m "Initial commit - pizza ordering system"
```

Step 3 - Create a GitHub repository

1. Go to **github.com** and sign in (or sign up - it is free).
2. Click the **+** in the top right -> **New repository**.
3. Name it **pizza-ordering-system**.
4. Leave it Public if you want to share it; Private if not.
5. Do **NOT** tick "Initialize with README" - we already have files.

6. Click **Create repository**.

Step 4 - Connect your local folder to GitHub

GitHub will show you commands. Use the ones for an existing repository:

```
git remote add origin https://github.com/YOUR_USERNAME/pizza-ordering-system.git
git branch -M main
git push -u origin main
```

On the first push, GitHub will ask for your password. You need to use a **Personal Access Token** instead - go to GitHub Settings -> Developer settings -> Personal access tokens -> Generate new token. Copy it and paste it as the password.

Step 5 - Daily workflow

Every time you change something:

```
git add .
git commit -m "Add cart feature"
git push
```

> **TIP**
>
> TIP: Write commit messages that explain *why*, not *what*. "Fix login" is better than "Update file". "Add price validation to place_order" is even better.

A good README.md template

Create **README.md** in the project root. It is the first thing anyone sees on GitHub:

```
# Pizza Ordering System

A simple pizza ordering web app built with PHP, MySQL, and Bootstrap 5.
Customers can register, browse pizzas, and place orders. An admin can
log in to approve them. Live updates via 5-second polling.

## Features
- Customer registration and login
- Two pizzas (Margherita, Pepperoni)
- Live customer dashboard
- Admin login and order approval
- Bootstrap 5 responsive UI

## Tech Stack
- PHP 8 + PDO
- MySQL 8
- Bootstrap 5.3
- Vanilla JavaScript (fetch + setInterval)

## Setup
1. Clone the repo into htdocs.
2. Import sql/schema.sql in phpMyAdmin.
3. Update config/db.php with your DB credentials.
4. Run make_hash.php once, insert the admin row, delete the file.
5. Visit http://localhost/pizza-app/

## Author
Your Name - github.com/yourusername
```

Chapter 15 - Testing, Deployment, and Where to Go Next

Testing checklist

Before you call the project done, walk through this list. Tick each item with [v] or [x] - if anything is [x], fix it before moving on.

Area	What to test
Registration	Can a new user sign up? Does duplicate email get rejected?
Login	Wrong password is refused? Right password gets in?
Sessions	Closing the browser and reopening - still logged in?
Menu	Both pizzas show? Prices are correct?
Order	Can you place an order? Quantity 0 is rejected?
Customer dashboard	New order appears? Status updates within 5s?
Admin login	Customer credentials are refused?
Admin dashboard	All orders show? Approve button works?
Logout	Both customer and admin logout work?
Direct URL access	Can you reach dashboard.php without logging in? (Should redirect)

Common bugs and how to fix them

Symptom	Likely cause	Fix
"Headers already sent"	Whitespace before <?php tag.	Remove blank lines / spaces above <?php.
Login form submits but nothing happens	session_start() missing.	Add session_start() at the top of header.php.
Cannot connect to database	Wrong host, username, or DB name in db.php.	Confirm credentials in phpMyAdmin.
Live update not working	JavaScript error blocks setInterval.	Open DevTools Console and read the red message.
Admin can log in as customer	Role check missing.	Add require_admin() at top of admin pages.

Deploying to a real server

When you are ready to put it online, you need:

- A shared hosting account (Hostinger, LankaHost, Namecheap, etc.) with PHP 8 and MySQL.

- A domain name (or use the free subdomain the host gives you).
- Access to cPanel - File Manager and phpMyAdmin.

Deployment steps:

1. Upload all project files to **public_html** via cPanel File Manager or FTP (FileZilla).
2. Create a MySQL database in cPanel and note the credentials.
3. Import **sql/schema.sql** via phpMyAdmin.
4. Edit **config/db.php** with the live DB credentials.
5. Run **make_hash.php** once on the live site, insert the admin row, delete the file.
6. Visit your domain - the site should work exactly like locally.

> **WARNING**
>
> WARNING: Always set **error_reporting(0)** on a live server - or set **display_errors = Off** in php.ini. Errors must be hidden from users; log them to a file instead.

Where to go next - extension ideas

Now that you have a working app, here are 10 features to add. Each one teaches a new concept:

1. **More pizzas** - add a real admin form to insert/edit/delete pizzas (full CRUD).
2. **Image uploads** - upload a pizza photo with each menu item.
3. **Shopping cart** - let customers add multiple pizzas before checkout.
4. **Order cancellation** - customers cancel pending orders.
5. **Email notifications** - send the customer an email when their order is approved.
6. **Search and filter** - search the menu by name or price range.
7. **Pagination** - show 10 orders per page on the admin dashboard.
8. **Order history graph** - admin sees a Chart.js bar chart of orders per day.
9. **Forgot password** - email a reset link with a one-time token.
10. **Convert to API + React** - keep the PHP backend, rebuild the frontend as a React single-page app.

Final words

You have built a complete web application from an empty folder. You have written the database schema, the backend code, the frontend, and added live updates. You have version-controlled it with Git and pushed it to GitHub. That is a portfolio project.

The next step is to **show it**. Add the GitHub link to your CV. Make a short video walkthrough and post it on LinkedIn. Write a blog post about what you learned. The project is only half the value - talking about it is the other half.

Good luck, and welcome to web development.

Quick Reference Card

File structure

```
pizza-app/
+-- config/db.php
+-- includes/
|   +-- header.php
|   +-- footer.php
|   `-- auth.php
+-- customer/
|   +-- register.php
|   +-- login.php
|   +-- logout.php
|   +-- menu.php
|   +-- place_order.php
|   +-- dashboard.php
|   `-- orders_api.php
+-- admin/
|   +-- login.php
|   +-- logout.php
|   +-- dashboard.php
|   +-- orders_api.php
|   `-- approve.php
+-- assets/style.css
+-- sql/schema.sql
+-- .gitignore
+-- README.md
`-- index.php
```

Key URLs (local)

Page	URL
Home	http://localhost/pizza-app/
Customer register	http://localhost/pizza-app/customer/register.php
Customer login	http://localhost/pizza-app/customer/login.php
Customer dashboard	http://localhost/pizza-app/customer/dashboard.php
Admin login	http://localhost/pizza-app/admin/login.php
Admin dashboard	http://localhost/pizza-app/admin/dashboard.php

End of tutorial

Built with care for beginners. Keep coding.