Smart Parking System

Technical Documentation

ESP32 Device • Next.js Website • Redis (Upstash)

Simulated test-mode payments with realistic card & UPI checkout

Complete technical reference for developers, collaborators, and reviewers. Includes architecture, full API reference, data model, source code appendix, deployment guide, and testing instructions.

Version 2.0

Contents

Executive Summary

The Smart Parking System automates entry, exit, and payment for a small parking facility with two monitored slots. An ESP32 microcontroller handles the physical side of the system — sensors, gate control, and an optional display — while a Next.js website provides a public status/payment page for drivers and an admin dashboard for the owner. A small Redis database bridges the two, since the website's serverless functions don't retain memory between requests.

Payments currently run in a fully simulated test mode: drivers see a realistic checkout with card and UPI options, but no real payment gateway is connected and no money changes hands. The two-step order/confirm design mirrors how a real gateway works, so integrating one later (e.g. Razorpay, Cashfree, Paytm) only requires changing a single file.

This document is organized as follows:

- **Sections 1–2** — system architecture and project layout.
- **Sections 3–4** — complete API reference and Redis data model.
- **Sections 5–7** — frontend pages, the payment flow, and ESP32 integration.
- **Sections 8–9** — environment variables and deployment steps.
- **Sections 10–11** — a testing guide and troubleshooting reference.
- **Section 12** — suggested next steps for a production rollout.
- **Appendix A** — the full source code of every backend file.

1. System Architecture

The system has three cooperating parts. Each is described below, followed by the full request/response cycle that ties them together.

1.1 ESP32 Device

The microcontroller reads an occupancy sensor for each of the two slots, tracks entry and exit times, computes the amount owed using pricing values it receives from the website, and drives the physical gate and any connected LCD. It holds no web pages of its own — its only network responsibility is to call the website's API roughly every two seconds.

1.2 Website (Next.js)

The website is a Next.js application deployed as serverless functions on Vercel, or as Netlify Functions via the Next.js plugin already configured in **netlify.toml**. It serves two pages — a public status/payment page and an owner admin dashboard — and exposes nine API routes covering device communication, public status, simulated payments, and admin operations.

1.3 Redis (Upstash)

Because serverless functions do not persist memory between invocations, all shared state lives in a small Redis database: live slot status, current pricing, pending payment confirmations awaiting device pickup, simulated order records, payment history, and admin login sessions. Upstash's free tier is sufficient for this workload.

1.4 End-to-End Request Flow

1. The ESP32 POSTs its current status to **/api/report** every ~2 seconds, including occupancy, computed price, gate state, and an acknowledgement of any previously applied payment.
2. A driver opens the website's public page, which polls **/api/status** every 2 seconds to show live slot information.
3. When a slot shows an amount due, the driver taps “Pay now,” opening a checkout modal with Card and UPI tabs.
4. The checkout calls **/api/pay** twice: first to create a simulated order for the slot's due amount, then to “confirm” it, which resolves to success roughly 80% of the time.
5. On a successful confirmation, the website writes a pending confirmation (**{ id, amount }**) into Redis for that slot, and logs the payment to history.
6. On its next poll, the ESP32 receives that pending confirmation, applies it (opening the gate), and echoes the confirmation's id back as an acknowledgement.
7. The website sees the matching acknowledgement on the next **/api/report** call and clears the pending record, completing the cycle.

2. Project Structure

The repository is a standard Next.js pages-router project. Below is the complete file layout, including files described elsewhere in this document.

```
smart-parking-website/
|-- lib/
|   |-- redis.js         Redis helpers: orders, slots, pricing, pending
|   |                     confirmations, payment history
|   `-- auth.js          Cookie-based admin session helpers
|-- pages/
|   |-- index.js          Public status page + test-mode payment checkout
|   |-- admin.js          Owner dashboard: login, pricing, history
|   `-- api/
|       |-- report.js     Device -> website status + pending confirmations
|       |-- status.js     Public read-only status, polled by index.js
|       |-- pay.js        Simulated payment: create order + confirm
|       `-- admin/
|           |-- login.js
|           |-- logout.js
|           |-- me.js
|           |-- setprice.js
|           `-- history.js
|-- netlify.toml          Netlify build config + Next.js plugin
|-- next.config.js        Next.js configuration
|-- package.json          Dependencies and scripts
`-- README.md             Setup and reference notes

parking_system_web_client.ino   ESP32 firmware (independent of the
                                  website's payment implementation)
```

3. Complete API Reference

Route	Method	Auth	Purpose
/api/report	POST	x-api-key header	Device sends live status, receives pricing + pending payment confirmation.
/api/status	GET	None (public)	Read-only slot status + pricing, polled by the website frontend.
/api/pay	POST	None (public)	Simulated payment: create order, then confirm (~80% approval).
/api/admin/login	POST	username + password	Verifies credentials, sets a session cookie.
/api/admin/logout	GET	Session cookie	Clears the session.
/api/admin/me	GET	Session cookie	Returns current login state.
/api/admin/setprice	POST	Session cookie	Updates base/rate/mintime/unit pricing.
/api/admin/history	GET	Session cookie	Returns the last 50 completed payments.

3.1 POST /api/report

Called by the ESP32 on every polling cycle. Requires the header **x-api-key: <DEVICE_API_KEY>**; requests without a matching key receive a 401.

```
Request body:
  {
    "now": "<device timestamp>",
    "slot1": { "occupied": bool, "price": number, "paid": bool,
               "entryTime": string, "exitTime": string, "gateOpen": bool },
    "slot2": { ...same shape as slot1... },
    "ack":  { "slot1": "<id>"|null, "slot2": "<id>"|null }
  }

Response body:
  {
    "pricing": { "base": number, "rate": number, "mintime": number, "unit": "sec"|"min"|"hr" },
    "pending": {
      "slot1": { "id": "<orderId>", "amount": number } | null,
      "slot2": { "id": "<orderId>", "amount": number } | null
    }
  }
```

3.2 GET /api/status

Public, read-only endpoint used by the website's own status page. Returns the same slot shape as above for both slots, defaulting to an empty/free state if the device has never reported, plus the current pricing object.

3.3 POST /api/pay

Drives the simulated payment flow in two steps. Step one creates an order for a given slot; step two confirms it, resolving to success roughly 80% of the time.

```
Step 1 - create order
  Request:  { "slot": 1 | 2 }
  Response: { "orderId": "<id>", "amount": number, "mode": "test" }

Step 2 - confirm order
  Request:  { "orderId": "<id>", "action": "confirm" }
  Response: { "orderId": "<id>", "status": "success" | "failed", "mode": "test" }
```

3.4 Admin Routes

All admin routes except **login** require a valid session cookie, set automatically by the browser after a successful login and checked against the **session:<token>** key in Redis. Requests without a valid session receive a 401.

```
POST /api/admin/login    { "username": string, "password": string } -> { "ok": true }
GET  /api/admin/logout   -> { "ok": true }
GET  /api/admin/me       -> { "loggedIn": bool, "username": string | null }
POST /api/admin/setprice { "base": number, "mintime": number,
                           "rate": number, "unit": "sec"|"min"|"hr" } -> { "ok": true }
GET  /api/admin/history  -> { "history": [ {slot, amount, entryTime,
                                            exitTime, paidAt, orderId}, ... ] }
```

4. Redis Data Model

All shared state is stored under a small number of predictable key patterns, summarized below.

Key	Shape	Notes
slot:1, slot:2	{occupied, price, paid, entryTime, exitTime, gateOpen, updatedAt}	Latest status reported by the device. Overwritten on every /api/report call.
pricing	{base, rate, mintime, unit}	Owner-adjustable via the admin page. Falls back to a sensible default if unset.
pending:slot:1 pending:slot:2	{id, amount}	Set when a payment is confirmed; cleared once the device acknowledges it. Expires after 10 minutes as a safety net.
order:<orderId>	{slot, amount, status}	Simulated payment order state: created, success, or failed. Expires after 1 hour.
history	list of {slot, amount, entryTime, exitTime, paidAt, orderId}	Completed, paid sessions, newest first — shown on the admin page (last 50).
session:<token>	true	Admin login session, checked via an HttpOnly cookie. Expires after 8 hours.

5. Frontend Pages

5.1 / — Public Status Page

The landing page shows both slots side by side, refreshing every 2 seconds via **/api/status**. Each slot card displays:

- Occupied / Free badge
- Entry and exit times
- Gate state (open/closed)
- Amount currently due
- A “Pay now (test mode)” button, shown only when a balance is due

Tapping the pay button opens the checkout modal described in Section 6. A footer link leads to the owner admin page.

5.2 /admin — Owner Dashboard

Gated behind a login form that checks the **ADMIN_USER** / **ADMIN_PASS** environment variables. Once authenticated, the dashboard offers:

- A pricing form to update the base fee, minimum included time, rate, and billing unit (seconds, minutes, or hours) — saved via /api/admin/setprice.
- A payment history list showing the most recent 50 completed transactions, each with slot, amount, and timestamp.
- A logout control that clears the session cookie.

6. Payment Flow (Test Mode)

Payments are fully simulated: there is no real payment gateway, no external API keys, and no money moves at any point. The design goal is a checkout that looks and feels like a real one, for demos, UI testing, and stakeholder review.

6.1 Checkout Experience

- Tapping “Pay now” opens a modal showing the slot number, amount due, and a “TEST MODE” indicator.
- Two tabs let the driver choose **Card** (card number, expiry, CVV, cardholder name) or **UPI** (a UPI ID such as name@bank).
- Client-side validation mirrors a real checkout: the card number must be 16 digits, expiry must match MM/YY, CVV must be 3–4 digits, and the UPI ID must match a plausible pattern.
- On submit, a ~1.4 second “Processing payment…” spinner plays before the result appears, matching the pacing of a real gateway.
- No card or UPI details ever leave the browser or reach the server — only the slot number and order ID are sent to /api/pay.
- The simulated result is approximately 80% success and 20% decline, matching the project's original simulated-payment logic.

6.2 Security Note

Because the card/UPI fields are cosmetic and never transmitted, there is no PCI-DSS or sensitive-data handling concern in this test-mode implementation. This changes once a real gateway is integrated — at that point, card data should be handled exclusively by the gateway's own hosted fields or SDK, never passed through this application's own backend.

6.3 Migrating to a Real Gateway

Only **pages/api/pay.js** needs to change. Its create-order/confirm shape mirrors how real gateways such as Razorpay, Cashfree, or Paytm operate, so the checkout modal, Redis helpers, and ESP32 firmware require no changes. The new implementation would:

1. Call the gateway's order-creation API instead of generating a local simulated order.
2. Replace the client-side card/UPI form with the gateway's hosted checkout or SDK.
3. Verify the gateway's payment signature/webhook server-side before marking an order successful.
4. Continue writing the same { id, amount } shape into pending:slot:<n> so the ESP32 integration is unaffected.

7. ESP32 Firmware Integration

File: **parking_system_web_client.ino**. This firmware required no changes when the payment method changed from a real gateway to the simulated test-mode flow, because the device never communicates with a payment gateway directly — it only calls **/api/report** and reacts to whatever **pending.slot1** / **pending.slot2** comes back in the response.

7.1 Responsibilities

- Reads the occupancy sensor for each slot.
- Computes the amount owed using the pricing values received from /api/report.
- Controls the physical gate and any connected LCD display.
- POSTs status and reads back pricing plus pending payment confirmations roughly every 2 seconds.
- Acknowledges an applied payment by echoing its id back in the next report's ack field.

7.2 Configuration Required Before Flashing

```
const char* SERVER_BASE_URL = "https://your-project.vercel.app"; // no trailing slash
const char* DEVICE_API_KEY  = "put-a-long-random-string-here";     // matches DEVICE_API_KEY
```

8. Environment Variables

Variable	Purpose
KV_REST_API_URL	Upstash Redis REST URL.
KV_REST_API_TOKEN	Upstash Redis REST token.
DEVICE_API_KEY	Long random string the ESP32 must send in the x-api-key header.
ADMIN_USER	Owner admin username.
ADMIN_PASS	Owner admin password.

No payment-gateway keys are required in the current setup — payments are simulated in test mode only.

9. Deployment Guide

9.1 Create a Free Redis Database

Visit console.upstash.com/redis, create a database, and copy the REST URL and REST token it provides.

9.2 Set Environment Variables

Add all five variables from Section 8 in your Vercel or Netlify project settings before the first deploy.

9.3 Deploy the Website

Vercel: push the project to a GitHub repository, then use “Import Project” in the Vercel dashboard. Next.js is auto-detected and no additional configuration is needed.

Netlify: the same repository can be imported directly; the Next.js plugin referenced in **netlify.toml** installs automatically during the build.

Either path produces a URL such as https://your-project.vercel.app.

9.4 Point the ESP32 at the Deployed Site

```
const char* SERVER_BASE_URL = "https://your-project.vercel.app";
const char* DEVICE_API_KEY  = "the same random string you put in env vars";
```

10. Testing Guide

The sections below describe how to manually verify each part of the system without needing the physical ESP32 hardware connected.

10.1 Simulating the Device

The device's role can be simulated with any HTTP client. For example, using curl:

```
curl -X POST https://your-project.vercel.app/api/report \
  -H "Content-Type: application/json" \
  -H "x-api-key: YOUR_DEVICE_API_KEY" \
  -d '{
    "now": "2026-08-01T10:00:00Z",
    "slot1": { "occupied": true, "price": 20, "paid": false,
               "entryTime": "09:45", "exitTime": "-", "gateOpen": false },
    "slot2": { "occupied": false, "price": 0, "paid": true,
               "entryTime": "-", "exitTime": "-", "gateOpen": true },
    "ack": { "slot1": null, "slot2": null }
  }'
```

A successful response includes the current pricing object and a pending field for each slot — both null until a payment has been confirmed.

10.2 Testing a Simulated Payment

With slot 1 occupied and unpaid as above, visit the public status page, wait for it to reflect that state, and tap “Pay now.” Complete either the Card or UPI form with any well-formatted values (no real card or account is needed) and submit. Roughly 4 out of 5 attempts will show “Payment approved.” Repeating a failed attempt is expected behavior, not a bug.

After a successful payment, re-run the curl command from Section 10.1 — the response's pending.slot1 field should now contain an id and amount, simulating what the real device would receive on its next poll.

10.3 Testing the Admin Dashboard

Visit /admin and sign in with the ADMIN_USER / ADMIN_PASS values configured in your environment. Confirm that updating the pricing form persists after a page refresh, and that any completed test payments appear in the history list.

11. Troubleshooting

Build fails with “Module not found” for a removed package

A leftover file (e.g. an old payment integration) is still importing a package no longer listed in package.json. Search the repository for the package name and remove or update the file that references it.

The device receives a 401 from /api/report

The x-api-key header sent by the ESP32 does not match the DEVICE_API_KEY environment variable configured on the deployed site. Confirm both values are identical, with no extra whitespace.

Admin login always fails

Double-check the ADMIN_USER and ADMIN_PASS environment variables are set on the deployment platform (not just locally), and that the deploy has picked up the latest values.

Pending payment never reaches the device

Confirm the device is actually polling /api/report and sending its ack field back correctly — the pending record is only cleared once the device's ack matches the stored id.

git rm -rf . hangs asking to retry a directory deletion

This is typically a locked file, often caused by working inside a cloud-synced folder (e.g. OneDrive). Pause syncing, close any open editors on those files, and retry.

12. Roadmap / Future Enhancements

1. **Real payment gateway** — replace the simulated logic in pay.js with a live integration once a business decision is made on which provider to use.
2. **Additional slots** — the current design supports two slots; extending to more would mean generalizing the slot1/slot2 fields into an array or numbered keys throughout the codebase.
3. **SMS/email receipts** — notify a driver's phone or email after a successful payment.
4. **Historical analytics** — expand the admin dashboard with charts of occupancy and revenue over time, beyond the current 50-entry history list.
5. **Multi-admin support** — move beyond a single shared ADMIN_USER/ADMIN_PASS pair toward per-user accounts if more than one person needs owner access.

Appendix A: Full Source Code Listing

The complete source of every backend file is included below for reference. Frontend page components (index.js, admin.js) are omitted from this listing for length, but are fully described in Section 5.

```
import { Redis } from "@upstash/redis";

// Reads KV_REST_API_URL / KV_REST_API_TOKEN from env (Upstash REST creds).
export const redis = new Redis({
  url: process.env.KV_REST_API_URL,
  token: process.env.KV_REST_API_TOKEN,
});

// ====================================================
//  ORDERS (simulated payments, created by /api/pay)
//  order:<orderId> -> { slot, amount, status: "created"|"success"|"failed" }
// ====================================================
export async function getOrder(orderId) {
  return redis.get(`order:${orderId}`);
}

export async function saveOrder(orderId, data) {
  return redis.set(`order:${orderId}`, data, { ex: 60 * 60 }); // expire in 1hr
}

// ====================================================
//  PENDING CONFIRMATIONS (website -> device)
//  pending:slot:<n> -> { id, amount }
//  The ESP32 polls /api/report and applies+acks this; once acked,
//  /api/report clears it.
// ====================================================
export async function setPendingConfirmation(slot, id, amount) {
  return redis.set(`pending:slot:${slot}`, { id, amount }, { ex: 60 * 10 });
}

export async function getPendingConfirmation(slot) {
  return redis.get(`pending:slot:${slot}`);
}

export async function clearPendingConfirmation(slot) {
  return redis.del(`pending:slot:${slot}`);
}

// ====================================================
//  LIVE SLOT STATUS (device -> website, via /api/report)
```

```
//  slot:<n> -> { occupied, price, paid, entryTime, exitTime, gateOpen, updatedAt }
// ======================================================
export async function setSlotStatus(slot, data) {
  return redis.set(`slot:${slot}`, data);
}

export async function getSlotStatus(slot) {
  return redis.get(`slot:${slot}`);
}

// ======================================================
//  PRICING (owner-adjustable via /api/admin/setprice)
// ======================================================
const DEFAULT_PRICING = { base: 10, rate: 1, mintime: 10, unit: "min" };

export async function getPricing() {
  const p = await redis.get("pricing");
  return p || DEFAULT_PRICING;
}

export async function setPricing(data) {
  return redis.set("pricing", data);
}

// ======================================================
//  HISTORY (completed + paid sessions, for the admin page)
//  history -> list of { slot, amount, entryTime, exitTime, paidAt, orderId }
// ======================================================
export async function addHistoryEntry(entry) {
  return redis.lpush("history", entry);
}

export async function getHistory(limit = 50) {
  return redis.lrange("history", 0, limit - 1);
}
```

```
import crypto from "crypto";
import { redis } from "./redis";

const COOKIE_NAME = "admin_session";
const SESSION_TTL_SECONDS = 60 * 60 * 8; // 8 hours

export function serializeCookie(token, maxAgeSeconds = SESSION_TTL_SECONDS) {
  const secure = process.env.NODE_ENV === "production" ? "; Secure" : "";
  return `${COOKIE_NAME}=${token}; HttpOnly; Path=/; Max-Age=${maxAgeSeconds}; SameSite=Lax${secure}`;
}

export function clearCookie() {
  return `${COOKIE_NAME}=; HttpOnly; Path=/; Max-Age=0; SameSite=Lax`;
}

export function getTokenFromReq(req) {
  const cookieHeader = req.headers.cookie || "";
  const match = cookieHeader.match(new RegExp(`${COOKIE_NAME}=([^;]+)`));
  return match ? match[1] : null;
}

export async function createSession() {
  const token = crypto.randomBytes(24).toString("hex");
  await redis.set(`session:${token}`, true, { ex: SESSION_TTL_SECONDS });
  return token;
}

export async function isValidSession(token) {
  if (!token) return false;
  const val = await redis.get(`session:${token}`);
  return !!val;
}

export async function destroySession(token) {
  if (token) await redis.del(`session:${token}`);
}
```

```
import {
  setSlotStatus,
  getPricing,
  getPendingConfirmation,
  clearPendingConfirmation,
} from "../../lib/redis";

// POST /api/report — called by the ESP32 every ~2 seconds.
// Requires header: x-api-key: <DEVICE_API_KEY>
//
// Body: { now, slot1: {...}, slot2: {...}, ack: { slot1, slot2 } }
// Response: { pricing: {...}, pending: { slot1: {id,amount}|null, slot2: ...} }
export default async function handler(req, res) {
  if (req.method !== "POST") {
    return res.status(405).json({ error: "Method not allowed" });
  }

  if (req.headers["x-api-key"] !== process.env.DEVICE_API_KEY) {
    return res.status(401).json({ error: "Invalid API key" });
  }

  const { now, slot1, slot2, ack } = req.body || {};

  if (slot1) await setSlotStatus(1, { ...slot1, updatedAt: now });
  if (slot2) await setSlotStatus(2, { ...slot2, updatedAt: now });

  // If the device's ack matches the pending id we sent, it means the
  // device has applied that payment — safe to clear it so we stop
  // resending it on future reports.
  let pending1 = await getPendingConfirmation(1);
  if (ack?.slot1 && pending1 && ack.slot1 === pending1.id) {
    await clearPendingConfirmation(1);
    pending1 = null;
  }

  let pending2 = await getPendingConfirmation(2);
  if (ack?.slot2 && pending2 && ack.slot2 === pending2.id) {
    await clearPendingConfirmation(2);
    pending2 = null;
  }
```

```
  const pricing = await getPricing();

  return res.status(200).json({
    pricing,
    pending: {
      slot1: pending1 || null,
      slot2: pending2 || null,
    },
  });
}
```

```
import { getSlotStatus, getPricing } from "../../lib/redis";

const EMPTY_SLOT = {
  occupied: false,
  price: 0,
  paid: true,
  entryTime: "-",
  exitTime: "-",
  gateOpen: true,
};

// GET /api/status — public, polled by the website's own frontend.
export default async function handler(req, res) {
  if (req.method !== "GET") {
    return res.status(405).json({ error: "Method not allowed" });
  }

  const [slot1, slot2, pricing] = await Promise.all([
    getSlotStatus(1),
    getSlotStatus(2),
    getPricing(),
  ]);

  return res.status(200).json({
    slot1: slot1 || EMPTY_SLOT,
    slot2: slot2 || EMPTY_SLOT,
    pricing,
  });
}
```

```
import {
  saveOrder,
  getOrder,
  getSlotStatus,
  setPendingConfirmation,
  addHistoryEntry,
} from "../../lib/redis";

// TEST-MODE ONLY. Simulates a payment gateway response with an ~80%
// approval rate — no real money, no external API calls, no keys required.
//
// Flow:
//   1. POST /api/pay { slot }               -> creates an order, returns { orderId, amount }
//   2. POST /api/pay { orderId, action:"confirm" } -> "runs" the simulated
//      payment and returns success/failure. On success, queues a
//      { id, amount } confirmation the device picks up on its next
//      /api/report poll, and logs a history entry for the admin page.

function randomOrderId() {
  return "SIM" + Date.now() + Math.floor(Math.random() * 10000);
}

export default async function handler(req, res) {
  if (req.method !== "POST") {
    return res.status(405).json({ error: "Method not allowed" });
  }

  const { slot, orderId, action, amount } = req.body || {};

  // Step 1: create a simulated order for a slot
  if (slot && !action) {
    if (slot !== 1 && slot !== 2) {
      return res.status(400).json({ error: "slot must be 1 or 2" });
    }

    let dueAmount = amount;
    if (dueAmount == null) {
      const status = await getSlotStatus(slot);
      dueAmount = status?.price ?? 20; // fallback test amount in INR
    }
```

```
    const newOrderId = randomOrderId();
    await saveOrder(newOrderId, { slot, amount: dueAmount, status: "created" });
    return res.status(200).json({ orderId: newOrderId, amount: dueAmount, mode: "test" });
  }

  // Step 2: "run" the simulated payment
  if (orderId && action === "confirm") {
    const order = await getOrder(orderId);
    if (!order) {
      return res.status(404).json({ error: "Order not found" });
    }

    const approved = Math.random() < 0.8; // ~80% approval

    await saveOrder(orderId, { ...order, status: approved ? "success" : "failed" });

    if (approved) {
      // The device reads this on its next /api/report poll and applies it.
      await setPendingConfirmation(order.slot, orderId, order.amount);

      // Log for the admin history page.
      const slotStatus = await getSlotStatus(order.slot);
      await addHistoryEntry({
        slot: order.slot,
        amount: order.amount,
        entryTime: slotStatus?.entryTime ?? "-",
        exitTime: slotStatus?.exitTime ?? "-",
        paidAt: new Date().toISOString(),
        orderId,
      });
    }

    return res.status(200).json({
      orderId,
      status: approved ? "success" : "failed",
      mode: "test",
    });
  }
```

```
  return res.status(400).json({ error: "Invalid request. Expected { slot } or { orderId, action:'confirm' }." });
}
```

```
import { createSession, serializeCookie } from "../../../lib/auth";

// POST /api/admin/login — { username, password }
export default async function handler(req, res) {
  if (req.method !== "POST") {
    return res.status(405).json({ error: "Method not allowed" });
  }

  const { username, password } = req.body || {};

  if (username !== process.env.ADMIN_USER || password !== process.env.ADMIN_PASS) {
    return res.status(401).json({ error: "Invalid credentials" });
  }

  const token = await createSession();
  res.setHeader("Set-Cookie", serializeCookie(token));
  return res.status(200).json({ ok: true });
}
```

```
import { getTokenFromReq, destroySession, clearCookie } from "../../../lib/auth";

// GET /api/admin/logout
export default async function handler(req, res) {
  const token = getTokenFromReq(req);
  await destroySession(token);
  res.setHeader("Set-Cookie", clearCookie());
  return res.status(200).json({ ok: true });
}
```

```
import { getTokenFromReq, isValidSession } from "../../../lib/auth";

// GET /api/admin/me — used by the admin page to check login state
export default async function handler(req, res) {
  const token = getTokenFromReq(req);
  const loggedIn = await isValidSession(token);
  return res.status(200).json({
    loggedIn,
    username: loggedIn ? process.env.ADMIN_USER : null,
  });
}
```

```
import { getTokenFromReq, isValidSession } from "../../../lib/auth";
import { setPricing } from "../../../lib/redis";

const VALID_UNITS = ["sec", "min", "hr"];

// POST /api/admin/setprice — { base, mintime, rate, unit } (requires login)
export default async function handler(req, res) {
  if (req.method !== "POST") {
    return res.status(405).json({ error: "Method not allowed" });
  }

  const token = getTokenFromReq(req);
  if (!(await isValidSession(token))) {
    return res.status(401).json({ error: "Not logged in" });
  }

  const { base, mintime, rate, unit } = req.body || {};

  if (base == null || mintime == null || rate == null || !unit) {
    return res.status(400).json({ error: "Missing fields: base, mintime, rate, unit" });
  }
  if (!VALID_UNITS.includes(unit)) {
    return res.status(400).json({ error: `unit must be one of: ${VALID_UNITS.join(", ")}` });
  }

  await setPricing({ base, mintime, rate, unit });
  return res.status(200).json({ ok: true });
}
```

```
import { getTokenFromReq, isValidSession } from "../../../lib/auth";
import { getHistory } from "../../../lib/redis";

// GET /api/admin/history — requires login
export default async function handler(req, res) {
  const token = getTokenFromReq(req);
  if (!(await isValidSession(token))) {
    return res.status(401).json({ error: "Not logged in" });
  }

  const history = await getHistory(50);
  return res.status(200).json({ history });
}
```

End of document. This reflects the project as of the current build — payments are simulated (test mode) with a realistic card/UPI checkout UI, and no real payment gateway is currently integrated.