THE FREE BUILDER COURSE · V4 · 2026 · A 2-WEEK PROGRAM

12 Systems, 14 Days, and the Code That Runs Them

The framework behind Tutoplus, BidLight, Tesla DGF, and Prigmar — distilled into twelve buildable systems, told in plain English, illustrated, paced as a two-week program, and paired with the working implementation from the Auto Showroom reference codebase.

Mouldi Nouri
Author & solo operator

2 weeks · 12 chapters
Read pace: ~1 chapter/day · finish in 14 days

free-builder.com
Get the framework

What's inside

Twelve chapters. One system per chapter. Two weeks end-to-end at a pace of one chapter a day, with a day off in the middle and a day to ship at the end. The story, the slides, the rules, and the working code that ships it.

01	**The Free Builder Framework**	Operating model: one operator, four pillars, an AI workforce
02	**Tech Stack — Why MERN**	JavaScript across the stack: solo-friendly, AI-friendly, free
03	**Infrastructure — Server, Domain & Deployment**	Rent the right server. Get on the internet. Ship a pipeline.
04	**SEO from Day One**	Compounding traffic starts before the product exists
05	**The Portal Layer — Next.js Portal + React Dashboard**	Two repos: Next.js marketing portal, React dashboard — both pulling their weight
06	**User Management**	Profiles, roles, authentication — the foundation customers stand on
07	**Permission Management**	Role-based access — who can see what, who can do what
08	**Project Management & Admin Tools**	The core domain CRUD — and the super-admin layer that runs the business
09	**Analytics & Measurement**	What you measure is what you improve
10	**The Core Service — Object & Flow**	What you actually sell — and the CRUD-plus-logic loop that delivers it
11	**Billing — Stripe Integration & Plan Gating**	Recurring revenue is the whole point. Wire it carefully.
12	**Growth Engine — Support, Marketing, Sales & Ops**	The flywheel that turns attention into revenue, on autopilot

CHAPTER 01 · THE FREE BUILDER COURSE

The Framework, or: How I Stopped Worrying and Started Shipping

An operator's manual disguised as a friendly chat. Four pillars, one person, an AI workforce, and a fair amount of duct tape.

BY MOULDI NOURI · SOLO SAAS BUILDER · A MAN WITH STRONG OPINIONS ABOUT MONGO

> *Welcome to the Free Builder framework. If you came here looking for "the one weird trick that built a $1M SaaS"… well, there are twelve of them. They're all weird. And they all work, more or less, depending on your taste in semicolons.*

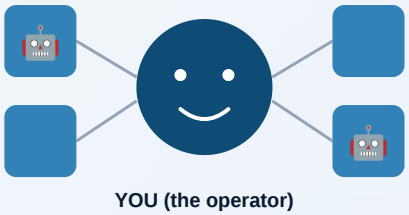
▸ HOW TO READ THIS BOOK

This is a **two-week program**. One chapter per day, six days a week. Day 7 is off (you'll need it). Day 14 you ship. By the end of fourteen days you will have read all twelve systems, watched the slides, read the production code, and have a working plan to put in motion the Monday after.

So what is this, exactly?

The Free Builder framework is what happened when I spent fifteen years building SaaS products solo, lost roughly four startups to "I'll figure out the marketing later", and finally distilled the survivors into a checklist I could hand my future self.

It is **not** a coaching package. It is not a "mindset". It is not a 40-hour video course where the first eight hours are someone explaining why they're qualified to teach the next eight hours. It's an **operating model** — twelve systems, four pillars, one operator (that's you), and a small army of AI workers doing the parts that don't deserve your attention.



One human in the middle, four AI workers doing the grunt work, all four pillars reporting in. This is the whole framework on one diagram.

The four pillars (the boring part that actually matters)

Every business — yours, mine, the corner pizza place — runs on four pillars whether anyone names them or not. Most solo builders only think about two of them, build half a product, and then wonder why nobody buys it. (Yes, I was that builder. Twice.)

- **Structure** — the technical spine. Stack, infra, auth, billing. The stuff that runs.
- **Service** — the thing customers actually pay you for. Your core service, delivered.
- **Marketing** — how strangers learn you exist. SEO, content, lead magnets, channels.
- **Sales** — turning attention into revenue. Funnels, follow-ups, closes.

Skip any one of them, and the business limps. Skip two, and you have a hobby. I am not here to be polite about this.

> 😅 AN ASIDE My first SaaS had a beautiful Structure pillar, a half-decent Service pillar, and basically no Marketing or Sales. I spent eight months perfecting a feature nobody asked for and then "launched"

by tweeting once. Total revenue: $0. Total lessons learned: priceless, though not as priceless as actual money would have been.

Who this is for (and who it isn't)

This is for the senior practitioner watching AI eat the commodity layer of their job and thinking, "I have a skill, I have ten years of pattern recognition, and I would like to keep this skill earning when the spreadsheet says it shouldn't." It's for the engineer with five abandoned side projects in their GitHub graveyard. It's for the design-savvy operator who can't ship without a developer co-founder.

It's **not** for someone looking for passive income that requires neither work nor judgment. If that's you, no shame, but please go find a different PDF. (May I recommend index funds.)

The honest filter. Three buckets. Be honest with yourself which one you're in.

The belief layer (mandatory; sorry)

I'd love to skip this part. It feels like the spinach in a meal that should be steak. But here we are: you cannot build a business solo without belief, and belief is built from four things.

- **Desire** — actually want this. Not "would be nice." Want.

- **A strong why** — at hour three of debugging a webhook signature at 1 AM, the why is the only thing that keeps you in the chair.
- **Self-trust** — knowing you'll do the thing you said you'd do.
- **Discipline** — the cousin of self-trust. Boring. Non-negotiable.

> *"Most people quit at the part where it gets boring. The framework is mostly the boring part." — me, every Sunday for a decade*

▸ KEY TAKEAWAY

You are not a developer. You are not a marketer. You are not a salesperson. You are **the operator** who connects all four pillars and points an AI workforce at each in sequence. Once you internalize that, the next eleven chapters write themselves.

The two middlewares the diagram doesn't show

Most "MERN architecture" diagrams I see online stop at `passport.authenticate` and call it a day. Real production apps add two more layers that earn their keep before week three:

- **`activityRecorder`** — a tiny middleware that runs after auth and writes one row to `activity_logs` for every authenticated request: `actor`, `method`, `path`, `status`, `durationMs`, `idempotencyKey`. Nine fields total. Doesn't block, fire-and-forget. The first time a customer asks "did Sarah delete that invoice last Tuesday?" you'll feel like a genius.
- **`requireActiveSubscription`** — runs after auth, before any business route. If the customer's subscription expired this morning, it returns `402 Payment Required` and a JSON envelope the frontend already knows

how to render as a billing modal. You're not running a free trial that turns into a forever-free tier by accident; you're enforcing the deal.

▸ THE WHOLE ORDER, IN 9 LINES

CORS → logger → JSON/urlencoded body parsers → cookie parser → session → passport init → `activityRecorder` → route mounts (each behind `passport.authenticate(...)` + `requireActiveSubscription`) → SPA fallback → 404 → error handler. Memorize this. Tape it to your monitor. Every other architectural decision flows from it.

What's coming up

Chapter 2 picks up the wrench. We're going to talk about why MERN — yes, that MERN, the one your CTO friend rolls their eyes at — is the right stack for solo SaaS in 2026. (Spoiler: it's not nostalgia. It's that AI workers know it cold.)

From there: infrastructure, the portal, users, permissions, billing, content, sales, ops, and the part where we talk about not burning out. Twelve chapters. Twelve systems. One operator. Eleven cups of coffee. Let's go.

LAUNCH, AUTOMATE, SCALE, YOUR SAAS SOLO.

Build a Fully Automated SaaS Business with the Free Builder Framework

WHO IS THIS COURSE FOR?

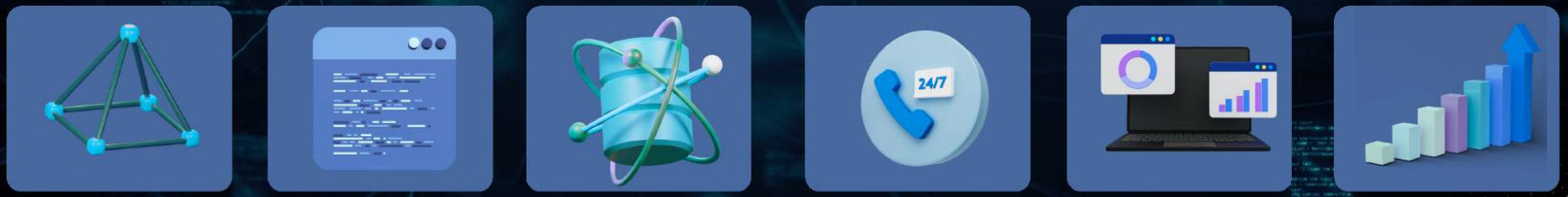
No more gatekeepers. No more waiting.

- Software engineer
- Indie hacker
- Employee starting a business

MADE FOR BUILDERS READY TO BREAK FREE

ALL THE HATS YOU'LL WEAR

You're Not Just a Dev/Marketer/Sales man—You're a Business Founder



- Infrastucture
- Devlopment
- Database
- Support
- Marketing
- Sales

KNOWING EACH ROLE IS THE KEY TO AUTOMATION.

WHY THIS WORKS?

Designed for Solo Builders by a Solo Builder

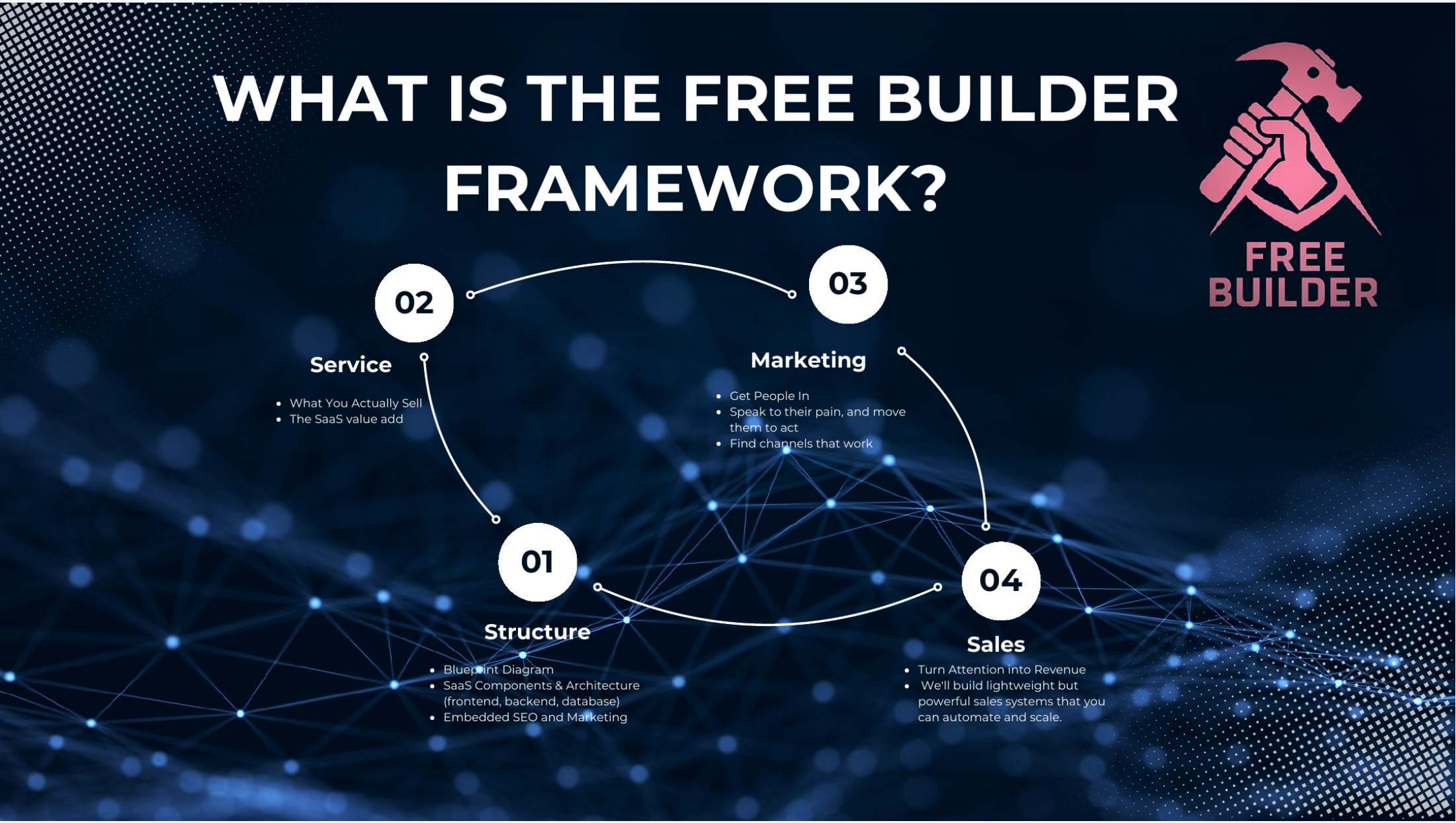
- 15 Years of SaaS
  - Built on the back of successful businesses & products

FREE BUILDER

Prigmar


THE BELIEF SYSTEM
4 blocks
Desire
Strong Why
Self-Trust
Discipline
SaaS is Built on Belief

WHAT IS THE FREE BUILDER FRAMEWORK?

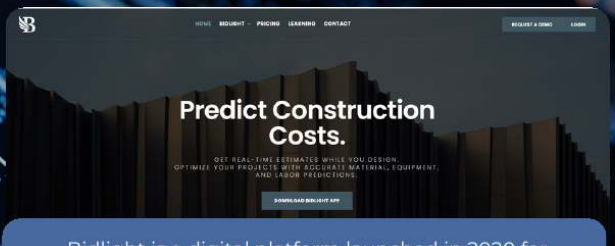


PROVEN BUSINESSES

What I Built with This Framework

TutoPlus is a 3D and VFX learning platform launched in 2012, built using many components of the Free Builder SaaS Framework.

The Agency is a 3D multiplayer shooter launched in 2017, built using many core components of the Free Builder SaaS Framework. Fast-paced, tactical, and visually immersive, it showcases the power and flexibility of the framework in a gaming environment.



Bidlight is a digital platform launched in 2020 for professionals in the architecture, engineering, and construction (AEC) industries. Bidlight offers real-time, AI-powered cost estimation tools that integrate seamlessly with Building Information Modeling software.

TESLA

Dgital Giga Factory (DGF) built in 2023.

Prigmar is a social media automation platform launched in 2025 to help brands and businesses grow organically through AI-driven content creation and scheduling.

COURSE STRUCTURE

Choose Your Path

Free	Deluxe (Free)
	SaaS Business Kit (App/Portal/Server Config files)
	Templates (emails, reachout ...)
Strategy & Systems	Strategy & Systems
Concepts & workflows	Concepts & workflows
Access to Community	Access to Community
1 Discovery call	1 Discovery call + 4 Coaching/follow-up sessions

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE back/app.js

The single file that wires the operator's spine. Read the order: security/CORS → body parsers → cookie/auth → request log → **activityRecorder** → resource routes (each behind `passport.authenticate` + **`requireActiveSubscription`**) → SPA → error handler. **Every** request that touches the system passes through this list. When a chapter mentions "the framework", it really means this ordering — change it and the rest of the architecture shifts.

```
// Express app factory. Wires middleware in the same order BidLight/Prigmar use:
// security/CORS → body parsers → session/auth → request log → routes → error handler.
const express = require('express');
const cors = require('cors');
const morgan = require('morgan');
const cookieParser = require('cookie-parser');
const path = require('path');

const config = require('./bin/config');
const activityLogs = require('./utils/activityLogs');

const app = express();

app.use(cors({ origin: config.corsOrigin, credentials: true }));
app.use(express.json({ limit: '1mb' }));
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(morgan(config.env === 'production' ? 'combined' : 'dev'));
app.use(activityLogs.requestRecorder());

// Mount routes — each route file is small, focused on one resource
app.use('/api/auth',          require('./routes/auth'));
app.use('/api/users',         require('./routes/users'));
app.use('/api/invites',       require('./routes/invites'));
app.use('/api/roles',         require('./routes/roles'));
app.use('/api/access-tokens', require('./routes/access-tokens'));
app.use('/api/cars',          require('./routes/cars'));
app.use('/api/leads',         require('./routes/leads'));
app.use('/api/billing',       require('./routes/billing'));
app.use('/api/analytics',     require('./routes/analytics'));
app.use('/api/admin',         require('./routes/adminPanel'));

// Health check (must be before the SPA catch-all)
app.get('/api/health', (req, res) => res.json({ ok: true, env: config.env }));

// Serve frontend build in production (after all /api routes so they take precedence)
```

```
if (config.env === 'production') {
  const buildDir = path.join(__dirname, '..', 'frontend', 'dist');
  app.use(express.static(buildDir));
  app.get(/^(?!\/api).*/, (req, res) => res.sendFile(path.join(buildDir, 'index.html')));
}

// Last-resort error handler
app.use((err, req, res, next) => {
  console.error('[error]', err.stack || err);
  res.status(err.status || 500).json({
    ok: false,
    error: err.expose ? err.message : 'Internal server error',
  });
});

module.exports = app;
```

DEMO FILE back/utils/activityLogs.js

The activity-recorder primitive. The middleware version is a 12-line wrapper around `CreateActivityLog(user, action, metadata)` shown here. Fire-and-forget — auth and audit must never block on logging. The schema captures actor/action/target/method/path/status/durationMs/idempotencyKey, which is the minimum you need when a customer asks "who did what, when?"

```
// Two helpers:
//   record({ actor, action, target, ... }) — fire-and-forget log a domain event.
//   requestRecorder() — middleware that attaches req.startedAt and logs slow/important requests.
const ActivityLog = require('../models/ActivityLog');

async function record({ actor, showroom, action, target, targetId, diff, req }) {
  try {
    await ActivityLog.create({
      actor,
      showroom,
      action,
      target,
      targetId,
      diff,
      ip: req ? (req.ip || req.connection?.remoteAddress) : null,
      ua: req ? req.get('user-agent') : null,
    });
  } catch (e) {
    // Logging must never break the request — swallow.
    console.warn('[activityLogs.record] failed:', e.message);
  }
}

function requestRecorder() {
  return (req, res, next) => {
    req.startedAt = Date.now();
    next();
  };
}

module.exports = { record, requestRecorder };
```

DEMO FILE frontend/src/store/index.js

The other half of the operator: a single Redux store plus thunk middleware for async actions. The store is the operator's *state*; the routes in app.js are the operator's *verbs*. Both halves run independently and communicate over a small REST surface — that's the whole architecture, end to end.

```
import { configureStore } from '@reduxjs/toolkit';
import { thunk } from 'redux-thunk';
import rootReducer from '../reducers';

const store = configureStore({
  reducer: rootReducer,
  middleware: (gdm) => gdm({ serializableCheck: false }).concat(thunk),
});

export default store;
```

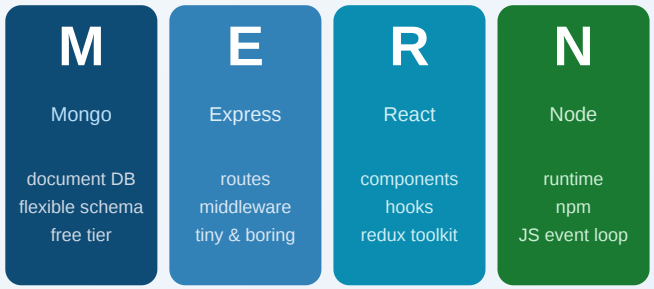
Why MERN, and Why I Stopped Arguing About It

JavaScript top to bottom — because your AI workers speak it natively and you only have one brain.

> *Every six months, someone on Twitter informs me that MERN is dead. Then I quietly ship another feature in three hours that would take their preferred stack three days, and I do not reply, because I am busy.*

The stack, in one boring paragraph

MERN = **M**ongo + **E**xpress + **R**eact + **N**ode. JavaScript everywhere. One language top to bottom. One package manager. One mental model. The shape that an LLM has seen approximately ten trillion times in its training data, which — and this is the part nobody likes — is the single biggest factor in how fast you'll ship in 2026.



Four lego bricks. The whole tower. I have shipped to four million dollars on these four bricks.

The unsexy reason it wins in 2026

I'll save you the religious debate. The honest case for MERN in 2026 is this:

- **AI training-data footprint.** Every model worth using was trained on a Mt. Everest of MERN code. Your AI worker will hallucinate less, ship faster, and fix its own bugs more reliably here than in any other stack.
- **One language.** Your brain has limited L1 cache. Switching between Ruby, SQL, Liquid, and Bash all day uses up working memory that should be going into your *service*.
- **Free tooling, free tier everywhere.** Mongo Atlas free tier, free Vercel, free Cloudflare. You can run a six-figure SaaS for under $30/month.
- **Hireable in a pinch.** When you finally outgrow solo, every freshly-laid-off engineer can read MERN by lunch.

> ⚡ HOT TAKE "Use the best tool for the job" is advice for people who have a team. Solo, you use the tool you already know cold, because the cost of context-switching to a "better" tool is paid by your only resource: your own time, daily.

What I am not telling you

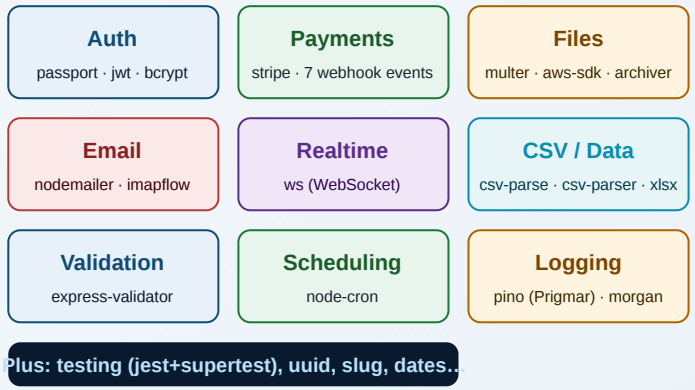
I am not telling you Mongo is the perfect database. (It isn't. Joins are awkward, transactions are a faff.) I am not telling you React is the most elegant framework. (It isn't. Solid is prettier.) I am telling you the stack you'll *actually ship on* is the one your AI workers know best, and that's MERN.

> ▸ KEY TAKEAWAY
>
> The best stack is the one your AI workforce can write fluently. In 2026, that's MERN by a wide margin. Pick it, stop reading "X vs Y" blog posts, and go ship.

The honest dependency count (the part nobody admits)

If you read a "Modern MERN starter" blog post, you'll see eleven backend deps and six frontend deps. Tidy. Lovely. Lies.

A real shipping SaaS has **30+ backend runtime deps** and well over a hundred on the frontend by year one. Not because anyone was sloppy — because production has needs that toy starters don't. Here's what actually shows up in BidLight and Prigmar:



Real production backend categories. None of these are optional — they're load-bearing for shipping customers.

The CSS confession

Now the awkward bit. The Modern MERN Starter says "no CSS-in-JS, no component library." Sounds noble. Both BidLight and Prigmar ship **Antd plus Bootstrap**, side by side, in the same bundle. (BidLight also has SASS,

because of course it does.)

This wasn't a planned choice. It's what happens when a deadline arrives, a designer hands you a screen with a date-picker on it, and Antd's date-picker is ten lines while the "pure" version is six hundred. The honest rule: **start with no component library**, then add Antd the day a real feature demands a widget that would take you a week to build. Don't add it before. Don't pretend you'll regret it later.

The testing and logging upgrades that earn their keep

Two libraries the toy starter skips that you'll want by month three:

- **jest + supertest** for backend tests. Hit your real routes with fake bodies, assert the response. Five minutes of setup, hours of saved 3 AM debugging. Prigmar runs ~80 tests in 12 seconds.
- **pino** for structured logging. Each log line is JSON; you grep production logs by `req_id` across hundreds of requests in a minute. Morgan is fine for dev — pino is what you want once paying customers exist.

▸ THE "TINY & BORING" REALITY CHECK

The starter is tiny and boring. The shipping app isn't. Both repos that built this framework — BidLight (AEC SaaS) and Prigmar (social-media SaaS) — sit around 30 backend deps and 100 frontend deps each. That's not a code-smell. That's the cost of paying customers.

WHY MERN POWERS THE FREE BUILDER FRAMEWORK?

It’s flexible, scalable, and free.

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE back/package.json

The cars-demo backend keeps the dep list small for teaching, but real apps (BidLight, Prigmar) grow this to 30+ runtime deps: `stripe`, `multer`, `nodemailer`, `aws-sdk`, `ws`, `express-validator`, `csv-parse`, `node-cron`, `pino` (logging), `jest` + `supertest` (testing). Every package is a future security fire — but skipping payments / file upload / email / realtime is also a fire. Pick which fires you want.

```
{
  "name": "auto-showroom-back",
  "version": "0.1.0",
  "private": true,
  "description": "Backend for Auto Showroom — Free Builder reference demo",
  "main": "bin/www",
  "scripts": {
    "start": "node bin/www",
    "dev": "nodemon bin/www"
  },
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "cookie-parser": "^1.4.6",
    "cors": "^2.8.5",
    "dotenv": "^16.4.5",
    "express": "^4.19.2",
    "express-session": "^1.18.0",
    "jsonwebtoken": "^9.0.2",
    "mongoose": "^8.4.0",
    "morgan": "^1.10.0",
    "stripe": "^15.0.0"
  },
  "devDependencies": {
    "nodemon": "^3.1.0"
  }
}
```

DEMO FILE frontend/package.json

Cars-demo frontend is the lean version. Real React dashboards (BidLight, Prigmar) ship ~100 deps including Antd + Bootstrap together (yes, both), chart libraries (ApexCharts, recharts), redux-persist, react-hook-form, FontAwesome, moment, html2canvas, jsPDF. The lesson: **start lean, add when a feature demands it**, never preemptively.

```
{
  "name": "auto-showroom-frontend",
  "version": "0.1.0",
  "private": true,
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview --port 5173"
  },
  "dependencies": {
    "@reduxjs/toolkit": "^2.2.0",
    "axios": "^1.6.0",
    "react": "^18.3.1",
    "react-dom": "^18.3.1",
    "react-redux": "^9.1.0",
    "react-router-dom": "^6.26.0"
  },
  "devDependencies": {
    "@vitejs/plugin-react": "^4.3.0",
    "vite": "^5.4.0"
  }
}
```

DEMO FILE back/app.js

Read this and notice how little is going on. Twenty lines of wiring, route mounts, error handler. That's the whole bootstrap. When engineers say "where's the magic?" — there is no magic. That's the magic. (The complexity, when it comes, lives in the routes and controllers, not the wiring.)

```
// Express app factory. Wires middleware in the same order BidLight/Prigmar use:
// security/CORS → body parsers → session/auth → request log → routes → error handler.
const express = require('express');
const cors = require('cors');
const morgan = require('morgan');
const cookieParser = require('cookie-parser');
const path = require('path');

const config = require('./bin/config');
const activityLogs = require('./utils/activityLogs');

const app = express();

app.use(cors({ origin: config.corsOrigin, credentials: true }));
app.use(express.json({ limit: '1mb' }));
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(morgan(config.env === 'production' ? 'combined' : 'dev'));
app.use(activityLogs.requestRecorder());

// Mount routes — each route file is small, focused on one resource
app.use('/api/auth',          require('./routes/auth'));
app.use('/api/users',         require('./routes/users'));
app.use('/api/invites',       require('./routes/invites'));
app.use('/api/roles',         require('./routes/roles'));
app.use('/api/access-tokens', require('./routes/access-tokens'));
app.use('/api/cars',          require('./routes/cars'));
app.use('/api/leads',         require('./routes/leads'));
app.use('/api/billing',       require('./routes/billing'));
app.use('/api/analytics',     require('./routes/analytics'));
app.use('/api/admin',         require('./routes/adminPanel'));

// Health check (must be before the SPA catch-all)
app.get('/api/health', (req, res) => res.json({ ok: true, env: config.env }));

// Serve frontend build in production (after all /api routes so they take precedence)
if (config.env === 'production') {
```

```
  const buildDir = path.join(__dirname, '..', 'frontend', 'dist');
  app.use(express.static(buildDir));
  app.get(/^(?!\/api).*/, (req, res) => res.sendFile(path.join(buildDir, 'index.html')));
}

// Last-resort error handler
app.use((err, req, res, next) => {
  console.error('[error]', err.stack || err);
  res.status(err.status || 500).json({
    ok: false,
    error: err.expose ? err.message : 'Internal server error',
  });
});

module.exports = app;
```

CHAPTER 03 · THE FREE BUILDER COURSE

Infrastructure: One VPS, A DNS Record, and the Courage to Press

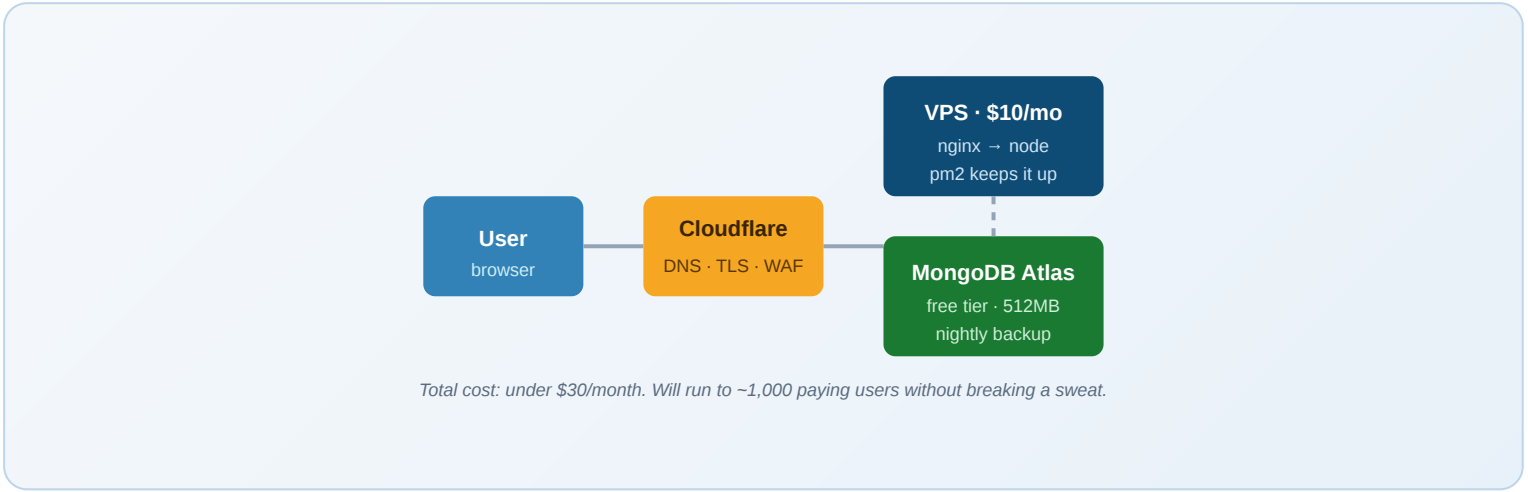
Enter

You don't need Kubernetes. You need a $10 server and a deploy script that fits on a postcard.

> *If your "infrastructure" requires a diagram, you're already losing. Solo infra is small enough to keep in your head, boring enough that nothing happens for months at a time, and cheap enough that your wife doesn't ask about it.*

The whole stack on one napkin

One VPS. One domain. One reverse proxy. One TLS cert. One database. One deploy pipeline. That's it. You are not Google. You will never be Google. Stop reading "scaling to a billion users" posts; you do not have a billion users; you do not even have ten.



The deploy pipeline that fits on a postcard

```
ssh you@your-vps "cd /var/www/app && \
  git pull && \
```

```
  npm install --omit=dev && \
  pm2 restart app"
```

Five lines. That is your CI/CD. You can put it in a Makefile, a npm script, or just memorize it. When the day comes that you have 100 paying customers and this stops being enough, you'll have plenty of revenue to upgrade.

> 😅 **TRUE STORY** I knew a guy who spent six weeks setting up Kubernetes for his side project. He never launched. He still hasn't. But the cluster is beautifully configured. It serves zero requests, very fast.

What I'm willing to spend money on

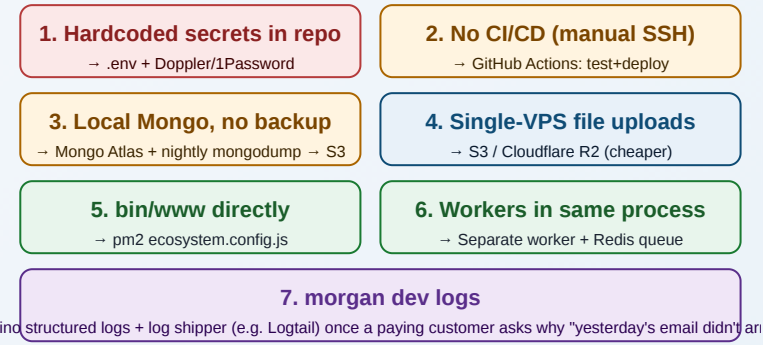
- **VPS** — Hetzner, DigitalOcean, OVH. $5–$20/month.
- **Domain** — Cloudflare registrar. Their margins are zero.
- **Database** — Mongo Atlas, free tier until you outgrow it.
- **Backups** — automated nightly mongodump to S3. About $0.30/month.
- **Monitoring** — Uptime Robot, free.

Everything else can wait. CI runners, container registries, observability stacks, log aggregators — those exist for companies with engineers, not companies with you.

> ▸ **KEY TAKEAWAY**
>
> Solo infra is a one-pager. VPS + Atlas + Cloudflare + a five-line deploy. Get it boring. Boring infra is the gift you give yourself.

The seven things you'll outgrow this on (and when)

The five-line deploy works. It will keep working. Until it doesn't. Here's the honest map of when each piece of this MVP-infra breaks and what replaces it. Bookmark this — you'll need it around month nine.



The seven graduation milestones. Don't pre-build any of them — but recognize the pain when it shows up.

The security confession

Both production codebases that built this framework — BidLight and Prigmar — committed their secrets to the repo for years. Plaintext API keys for Stripe, S3, SMTP, Calendly, OAuth providers across 11 social platforms. It's the kind of thing you'd fail an audit on, and the kind of thing nobody fixes until "we got pwned" turns up in a Slack thread.

The fix is boring: **extract every `config.js` string into `.env`**, never commit `.env`, use a secret manager (Doppler / 1Password / AWS Secrets Manager) in production, and have `.env.example` be the only file in the repo. Make this Day-1 hygiene, not a year-three crisis.

> 🔥 **THE AUDIT STORY** A friend was preparing for SOC 2 and ran `git log -p | grep -i "key\|secret\|password"` on his repo. Got 847 hits. He cried. Then he rotated everything and added a pre-commit hook. The pre-commit hook is now in *my* repo, because I'm not crying twice.

One more thing: backups

You will not implement backups until the first time you lose data, after which you will become a backup zealot. Skip the trauma. Add this to crontab tonight:

```
0 3 * * * mongodump --uri="$MONGO_URI" --archive=/tmp/db-$(date +\%F).gz --gzip && \
  aws s3 cp /tmp/db-$(date +\%F).gz s3://your-backups/ && \
  rm /tmp/db-$(date +\%F).gz
```

30 days S3 lifecycle policy, ~$0.30/month. The day you accidentally `dropCollection` in production at 4 PM on a Friday, this line of crontab is worth twenty years of your earning potential.

> ▸ **KEY TAKEAWAY, EXPANDED**
>
> Solo infra is a one-pager. *And* it's a one-pager with seven known graduation points. Build the boring version first. When it breaks, you'll know exactly which knob to turn.

SERVER RENTING, DEVELOPMENT & DEPLOYMENT

Start configuring your environment. This is your service/product's home.

- Git Setup

- Rent a server

- Install Protocols

- Start configuring your environment.

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE `back/bin/config.js`

Every config value reads from `process.env` once, in this file. No other module touches env directly. The cars-demo version is clean; the production parents (BidLight, Prigmar) committed plaintext secrets here for years — graduation move #1 is extracting everything into `.env` and a secret manager.

```
// Central env loader. Every other module reads from here, never from process.env directly.
require('dotenv').config();

const config = {
  port: parseInt(process.env.PORT || '4000', 10),
  env: process.env.NODE_ENV || 'development',
  mongoUri: process.env.MONGODB_URI || 'mongodb://localhost:27017/auto-showroom',
  jwtSecret: process.env.JWT_SECRET || 'dev-secret-change-me',
  stripeSecret: process.env.STRIPE_SECRET_KEY || '',
  corsOrigin: process.env.CORS_ORIGIN || 'http://localhost:5173',
};

if (config.env === 'production' && config.jwtSecret === 'dev-secret-change-me') {
  console.warn('[config] JWT_SECRET is using the dev default in production. Set it in .env.');
}

module.exports = config;
```

DEMO FILE back/bin/db.js

Mongoose connection with an in-memory fallback so the app boots even when Mongo is down. The fallback is intentionally *just enough* — it logs a warning and keeps the process alive. The real safety net is your nightly Mongo backup, not in-process state.

```
// Mongoose connection. If MongoDB is unreachable, falls back to in-memory mode so the
// demo still boots — the frontend will see real-looking API responses backed by an
// in-process store. Production should always use a real Mongo URI.
const mongoose = require('mongoose');
const config = require('./config');

let connected = false;
const memStore = new Map();

async function connect() {
  if (connected) return mongoose.connection;
  try {
    await mongoose.connect(config.mongoUri, { serverSelectionTimeoutMS: 2500 });
    connected = true;
    console.log('[db] connected to', config.mongoUri);
    return mongoose.connection;
  } catch (err) {
    console.warn('[db] Mongo unavailable — falling back to in-memory store. Reason:', err.message);
    return null;
  }
}

function getMemStore() {
  return memStore;
}

module.exports = { connect, getMemStore, isConnected: () => connected };
```

DEMO FILE back/bin/www

Process entry point. Three lines of business logic: load config → connect DB → start HTTP server. Anything more belongs in app.js. The boringest file in the codebase, by design. Graduation move #5: wrap it with a pm2 `ecosystem.config.js` when you want zero-downtime restarts.

```
#!/usr/bin/env node
// Process entry — boots Express, opens DB, starts listening.
const http = require('http');
const config = require('./config');
const { connect } = require('./db');
const app = require('../app');

(async () => {
  await connect();
  const server = http.createServer(app);
  server.listen(config.port, () => {
    console.log(`[www] Auto Showroom API listening on :${config.port} (${config.env})`);
  });
})();
```

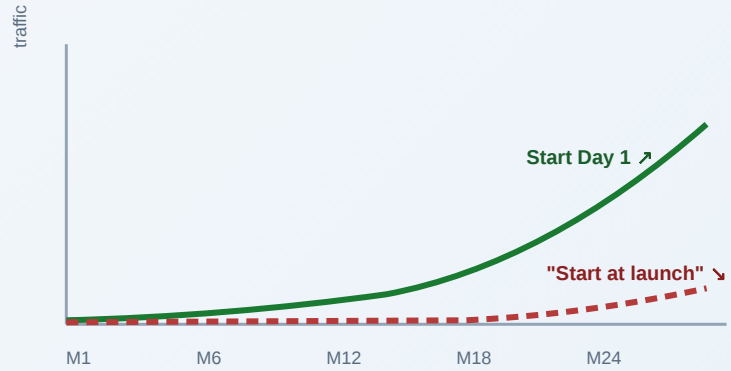
SEO from Day One, Because Tomorrow Is Already Late

The compounding lever every solo SaaS forgets to pull until it's too late to matter.

> *SEO is the single highest-leverage thing a solo SaaS operator can do, and it is also the single thing every solo SaaS operator decides to "get to next quarter". Don't be that operator. Be the other one.*

The cruelty of compound interest

Here is the unfair part: every blog post you publish today will still be earning you traffic two years from now. Every Tweet — even the viral ones — earns you traffic for about 36 hours and then disappears like a sneeze.
Compounding works the same way for content as it does for money. Start late, you stay late.



Two operators. Same product. Same launch date. One published on Day 1; one waited for launch. After 24 months, this is the gap.

The three lazy moves that work

You do not need to be a content marketer. You need to do three lazy things consistently for nine months.

- **Define your target queries** — the actual phrases your future customer types when they're frustrated. "How to organize used car inventory in Excel" is a real query. Your product is the answer.
- **Write one post per week** that answers one query end-to-end. Not *about* the query. *Answering* it. Your AI worker can draft these in 20 minutes. You spend 40 minutes editing in *your* voice.
- **Every post ends with a CTA** — newsletter or product, your choice. No CTA is a wasted post. (Yes, every single one. I will fight you on this.)

> 📣 **PSA** If your blog has zero CTAs, you are not running a blog, you are running a hobby. There is no shame in a hobby. Just don't confuse them.

The five-minute SEO checklist

- Title tag = the query, almost verbatim
- One H1 per page; H1 contains the query
- Meta description, 150 chars, makes the click feel earned
- Internal links to two other relevant posts
- One image, with an alt tag that's also a query you'd want to rank for

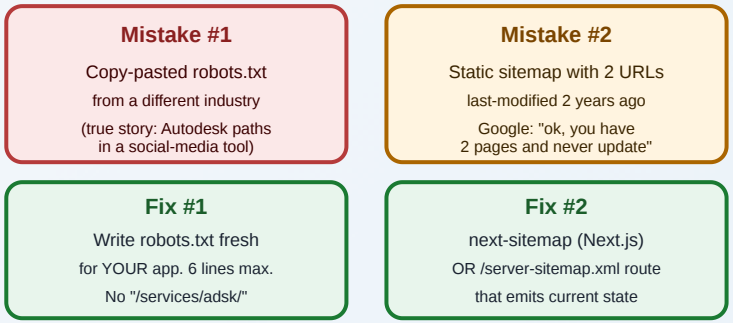
That's all you need. Schema.org markup, Open Graph, sitemaps — bolt them on later. The first nine months are about *publishing*, not optimizing the eleventh percentile.

▸ KEY TAKEAWAY

Publishing on Day One compounds into a moat that no amount of paid advertising can outrun. The bar is one post a week, written in your voice, with a CTA. Boring, repeatable, devastating.

The robots.txt and sitemap mistakes everyone makes

I've now reviewed two production codebases (BidLight and Prigmar) where the team got SEO almost right and one detail almost-comically wrong. **Don't be them.**



Two boring fixes. Five minutes apiece. Will save you from looking like an amateur in front of Google.

The OG tag bug you'll commit twice (and a fix)

Most apps end up with one of two problems on Open Graph tags:

- The tags are **commented out** in `index.html` "to come back to later" (BidLight: yes, this happened).
- The tags are **set once** in `index.html`, so the homepage, pricing page, blog post, and dashboard all share the same OG image and description (most SPAs).

The fix: a single `Seo` component (like Prigmar's `components/seo/Seo.js`) that you wrap every page with. It accepts `title`, `description`, `ogImage`, and writes them into `<Head>` via Next.js or react-helmet on a CRA. Every page becomes one line of OG hygiene.

The graduation tier: Schema.org and a real CMS

Once your blog has 30+ posts and is bringing in organic visitors, two upgrades become worth it:

- **JSON-LD `Article` schema** on every blog post — author, datePublished, image, headline. Adds the rich-snippet stars in Google results. ~15 lines per post template.
- **A real CMS, or at least an editorial workflow.** Prigmar has `SeoManagement` + `seoPublishTargets`: in-app editor for SEO articles, then publish-target adapters that push to WordPress / Ghost / your own backend. The setup paid for itself in week three.

> 🤦 **STORY TIME** I reviewed an app that copy-pasted its robots.txt from a CAD/BIM tool. The new app was a social-media scheduler. Their robots.txt disallowed `/services/adsk/c/` and `/redshift/search/`, neither of which existed. Google didn't mind — Google doesn't care about phantom disallows — but the next engineer who read the file lost ten minutes wondering if they'd joined an Autodesk subsidiary by mistake.

▸ KEY TAKEAWAY, EXPANDED

Day-1 SEO is title, H1, meta description, and one post a week. Month-6 SEO is a per-page `Seo` component and a real sitemap. Year-1 SEO is JSON-LD and a publishing workflow. Add each layer when the previous one is full.

START MARKETING DAY 1 – THE SEO ENGINE

This is what will bring you leads in the long run!

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE `frontend/index.html`

Four lines that matter for SEO: `<title>`, `<meta name="description">`, `<meta name="viewport">`, and the `lang` attribute. Cars-demo keeps it minimal on purpose. The graduation move is a per-page `Seo` component (covered in Ch 5) — but until you have 10 pages, this single index.html is enough.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" href="/favicon.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta name="description" content="Auto Showroom — buy and sell quality used cars. Live inventory, easy financing, 7-day
guarantee." />
    <title>Auto Showroom — Cars worth driving</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

CHAPTER 05 · THE FREE BUILDER COURSE

The Portal Layer, or: Why Your Marketing Site Should Not Be Your Product

Two surfaces. Two codebases. One auth boundary. Stop trying to do it all in one bundle.

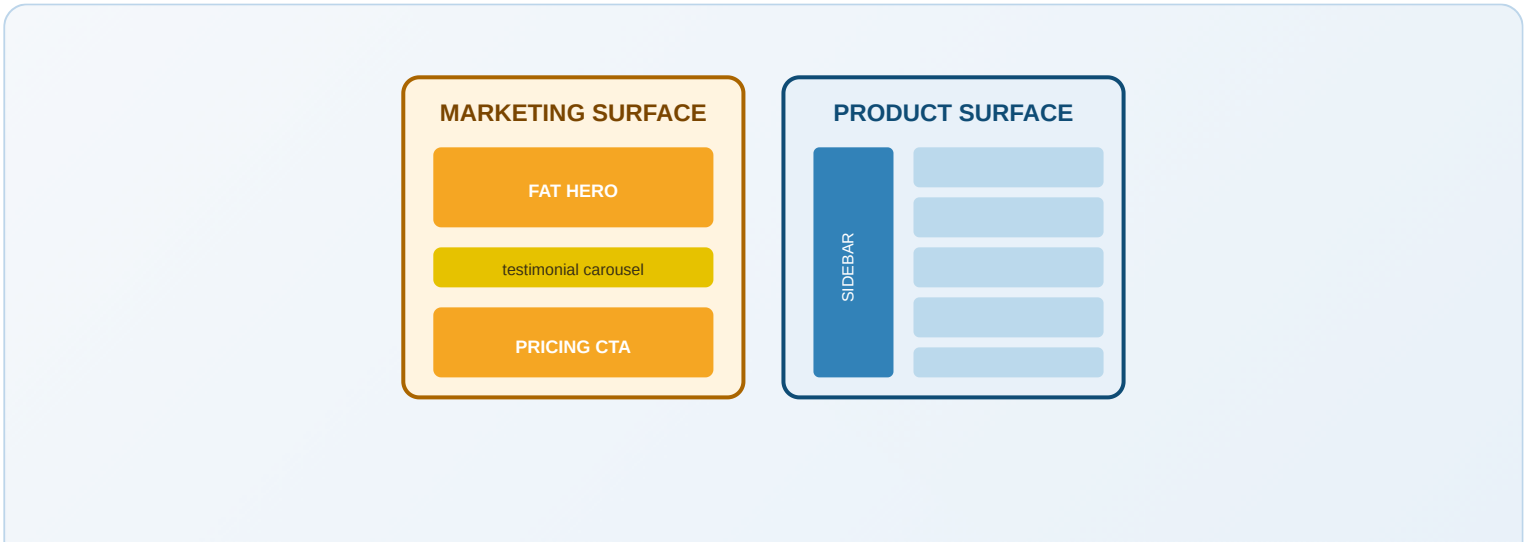
> *Half of all solo SaaS projects I've reviewed have a single confusing mass where the marketing site and the product app are tangled into one React tree. They render the same Header on the pricing page and the dashboard, and they wonder why their Lighthouse score is 38.*

Two surfaces, one operator

There are two completely different audiences hitting your domain:

- **Strangers** — Googled their way in, never seen you before, want to know in eight seconds whether you're worth a click. They are on the marketing surface.
- **Customers** — already paid, logged in, doing real work. They are on the product surface.

These two surfaces have nearly opposite design goals. Marketing wants fat hero images, generous whitespace, animated testimonials. Product wants tight density, fast tab switches, zero pixels wasted. Trying to serve both in one component tree is the developer equivalent of arguing with your spouse via a microwave.



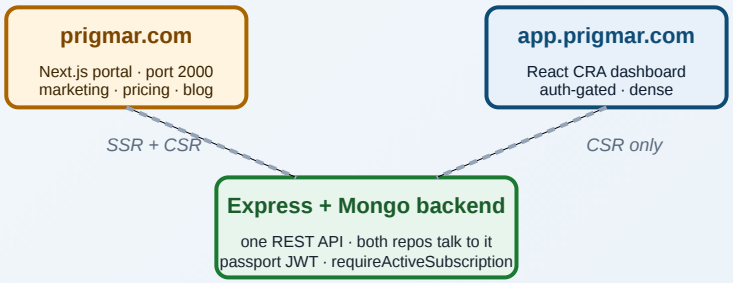
Same domain, same codebase. Two entirely different products. Stop trying to make them look the same.

The honest answer: TWO codebases, one auth boundary

Earlier drafts of this chapter pitched "one codebase, two Next.js layout groups." Sounds elegant. But after auditing two real production stacks (BidLight and Prigmar), the truth is uglier *and* better: **two separate repos talking to the same backend**.

- **`prigmar-portal`** — Next.js 12 (Pages Router), runs on its own systemd unit at port 2000, serves `prigmar.com` (marketing, pricing, blog, Q&A, contact).
- **`prigmar-app`** — React CRA SPA, served by the Express backend at `app.prigmar.com` (dashboard, settings, every authenticated screen).

Both repos call the *same* backend over REST (`NEXT_PUBLIC_SERVER_URL=https://app.prigmar.com`). The portal stays small (~600 files, no API routes, no business logic) and is optimized for SEO and speed; the dashboard is big and dense and optimized for power users.



Two repos, two deploys, one auth boundary. Marketing optimized for SEO. Dashboard optimized for power users. Both call the same REST API.

Why two repos beats one

- **Build speed.** Marketing pages bundle separately. Adding a chart library to the dashboard doesn't bloat your pricing-page bundle.
- **Cache headers.** The portal can be aggressively cached at the CDN; the dashboard can't. Different cache rules at different vhosts.
- **SSR / SSG ergonomics.** Next.js's `getServerSideProps` and (when you graduate to App Router) RSCs work properly when the portal is its own app. Trying to do SSR inside a CRA dashboard is a recipe for bedtime tears.
- **Different teams, eventually.** Marketing writers, designers, and one engineer can ship to the portal without ever touching the product. That separation is worth its weight in unmerged PRs.

The portal anatomy (what's actually in a real Next.js marketing site)

If you've never built one, here's what the Prigmar portal ships — call it your shopping list:

- **30 routes** under `pages/`: home, pricing, how-it-works, features, services, articles (with `[slug]` SSR), questions/Q&A, news, users, contact (Calendly embed), login, signup, auth.
- **A reusable `Seo` component** wraps every page's `<Head>` — title, description, OG, Twitter cards, canonical. The single most-imported component in the codebase.
- `next-sitemap` in the post-build step generates `sitemap.xml` + `robots.txt` from the route table. Setup: 15 lines of config. Maintenance: zero.
- **A pricing test suite** — pure functions for `getDisplayPrice`, `getAnnualSavings`, `decideCheckoutAction` in `utils/pricing.js`, with jest covering each. Pricing math has a way of being wrong in ways the QA team won't notice; tests are the cheapest insurance.
- **GA4 + custom event tracking** wired in `_app.js`, so every page view fires an analytics event with the user ID (if logged in) or anonymous ID otherwise.
- **One Calendly embed** on the contact page that doubles as a demo-request form — submits to the backend's `/api/demo/request` endpoint then opens the calendar.

The CSR-marketing trap (and how to spot it)

One thing the Prigmar portal still gets wrong: most marketing pages are client-rendered. `useEffect` + axios on the homepage. Open the Network tab in DevTools and you'll see an empty HTML shell followed by a flurry of API calls. Google can crawl this — barely — but it's slower than it should be and the first-byte SEO score takes the hit.

The fix is small. Convert each marketing page to `getStaticProps` with a 60-second revalidate. The page becomes pre-rendered HTML at build (or on-demand at request), then becomes interactive on the client. Same UI, dramatically better Lighthouse. One file at a time.

> 📈 **EASY WIN** Last app I audited went from a 41 Lighthouse score to 96 by converting four marketing pages from CSR to SSG. Total time: an afternoon. Total impact: more organic signups in week two than the previous quarter. Sometimes the engineering payoff is so silly it feels illegal.

The protected-route pattern (still six lines)

On the dashboard side (the React CRA, which IS what cars-demo ships), `App.jsx` has a six-line `ProtectedRoute` component that gates every authenticated route on a role check. It's not framework magic, it's a function. Open it, read it, copy it, ship it. You do not need a 200-line auth library; you need six lines and a Redux selector.

The "no BidLight portal" caveat

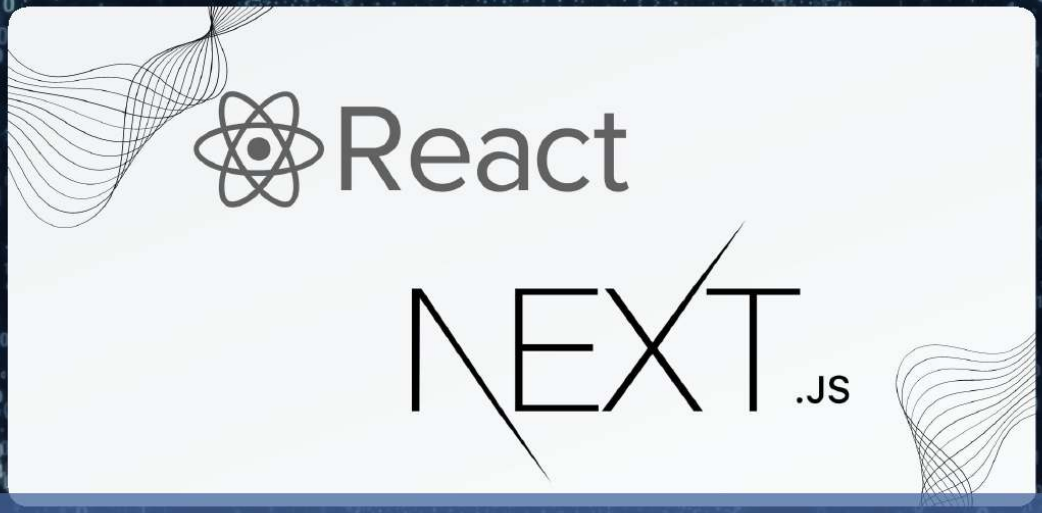
You'll notice this chapter draws from Prigmar's portal codebase only. BidLight (the other production app behind this framework) doesn't have a separate marketing portal — its landing page is just another route in the dashboard SPA. That's a real choice and you might make it too: when your audience is enterprise B2B and discovery happens through sales conversations rather than Google, the portal layer earns less. **Build it when SEO is your acquisition channel. Skip it when sales calls are.**

▸ **KEY TAKEAWAY, EXPANDED**

Marketing and product are two different products that happen to share a backend. Two repos, two deploys, one REST API. Next.js for the portal, React for the dashboard. Convert your marketing pages

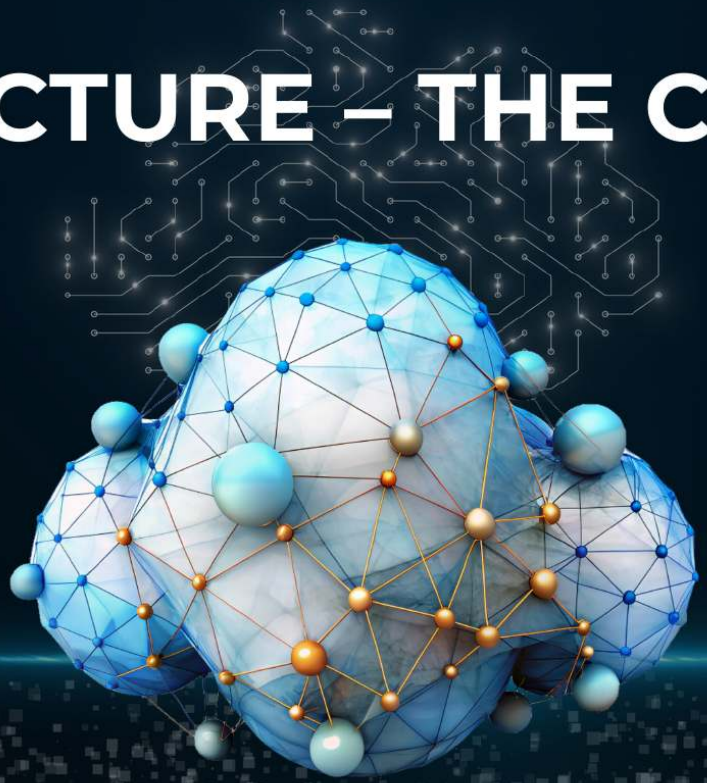
to SSG. Use a reusable `Seo` component. `next-sitemap` for the sitemap. Done.

WHY NEXTJS POWERS THE PORTAL SYSTEM?



Search engines friendly! fast rendering? more exposure ..

SAAS STRUCTURE – THE CORE ENGINE



Structure is the engine that runs everything else

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE frontend/src/App.jsx

Cars-demo ships the dashboard half — React CRA with react-router. The six-line `ProtectedRoute` wrapper gates every authenticated route on a role check, with a graceful loading state when a persisted token is still resolving. The portal half (Next.js) lives in a separate repo (`prigmar-portal-development`) — this chapter pairs the two patterns; cars-demo currently only ships the dashboard side.

```
import React, { useEffect } from 'react';
import { Routes, Route, Navigate } from 'react-router-dom';
import { useDispatch, useSelector } from 'react-redux';
import { loadMe } from './actions/authActions';

import Header from './components/Header';
import Home from './pages/Home';
import Inventory from './pages/Inventory';
import CarDetail from './pages/CarDetail';
import Login from './pages/Login';
import Signup from './pages/Signup';
import ForgotPassword from './pages/ForgotPassword';
import ResetPassword from './pages/ResetPassword';
import VerifyEmail from './pages/VerifyEmail';
import AcceptInvite from './pages/AcceptInvite';
import Profile from './pages/Profile';
import Dashboard from './pages/Dashboard';
import Admin from './pages/Admin';
import AdminUsers from './pages/admin/Users';
import AdminInvites from './pages/admin/Invites';
import AdminRoles from './pages/admin/Roles';
import ApiKeys from './pages/account/ApiKeys';
import Docs from './pages/Docs';

function ProtectedRoute({ children, roles }) {
  const { user } = useSelector(s => s.auth);
  // While loadMe() is still resolving the persisted token, render a loading shell
  // instead of bouncing to /login. The token's presence tells us a session is in-flight.
  const hasPersistedToken = typeof window !== 'undefined' && !!localStorage.getItem('auth_token');
  if (!user) {
    if (hasPersistedToken) return <div className="section narrow"><p className="muted">Loading…</p></div>;
    return <Navigate to="/login" replace />;
  }
  if (roles && !roles.includes(user.role) && user.role !== 'admin') return <Navigate to="/" replace />;
  return children;
}
```

```
export default function App() {
  const dispatch = useDispatch();
  useEffect(() => { dispatch(loadMe()); }, [dispatch]);
  return (
    <>
      <Header />
      <main className="app-main">
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/inventory" element={<Inventory />} />
          <Route path="/cars/:id" element={<CarDetail />} />
          <Route path="/docs" element={<Docs />} />

          <Route path="/login" element={<Login />} />
          <Route path="/signup" element={<Signup />} />
          <Route path="/forgot-password" element={<ForgotPassword />} />
          <Route path="/reset-password/:token" element={<ResetPassword />} />
          <Route path="/verify-email/:token" element={<VerifyEmail />} />
          <Route path="/accept-invite/:token" element={<AcceptInvite />} />

          <Route path="/dashboard" element={<ProtectedRoute><Dashboard /></ProtectedRoute>} />
          <Route path="/profile" element={<ProtectedRoute><Profile /></ProtectedRoute>} />
          <Route path="/account/api-keys" element={<ProtectedRoute><ApiKeys /></ProtectedRoute>} />

          <Route path="/admin" element={<ProtectedRoute roles={['manager','admin']}><Admin /></ProtectedRoute>} />
          <Route path="/admin/users" element={<ProtectedRoute roles={['manager','admin']}><AdminUsers /></ProtectedRoute>} />
          <Route path="/admin/invites" element={<ProtectedRoute roles={['manager','admin']}><AdminInvites /></ProtectedRoute>} />
          <Route path="/admin/roles" element={<ProtectedRoute roles={['admin']}><AdminRoles /></ProtectedRoute>} />

          <Route path="*" element={<Navigate to="/" replace />} />
        </Routes>
      </main>
      <footer className="app-footer">
        <p>Auto Showroom — a Free Builder reference demo. <a href="/docs">Docs</a> · <a href="https://free-builder.com/">free-
builder.com</a></p>
      </footer>
    </>
  );
}
```

DEMO FILE `frontend/src/components/Header.jsx`

One header, role-aware. Notice the role guards on the admin links — they appear only when the user can actually use them. This is the cheapest UX upgrade in the codebase and the most ignored one in tutorials. (Note: the portal repo has its own marketing header at `components/layout/header/` — same idea, different visual language.)

```
import React from 'react';
import { Link, NavLink } from 'react-router-dom';
import { useSelector, useDispatch } from 'react-redux';
import { logout } from '../actions/authActions';

export default function Header() {
  const { user } = useSelector(s => s.auth);
  const dispatch = useDispatch();
  const isStaff = user && (user.role === 'admin' || user.role === 'manager' || user.role === 'sales');
  return (
    <header className="app-header">
      <Link to="/" className="logo">Auto<span>Showroom</span></Link>
      <nav className="nav">
        <NavLink to="/inventory">Inventory</NavLink>
        {user && <NavLink to="/dashboard">Dashboard</NavLink>}
        {isStaff && <NavLink to="/admin">Admin</NavLink>}
        <NavLink to="/docs">Docs</NavLink>
      </nav>
      <div className="auth-cluster">
        {user ? (
          <>
            <Link to="/profile" className="hello">Hi, {user.name || user.email}</Link>
            <button className="btn-ghost" onClick={() => dispatch(logout())}>Sign out</button>
          </>
        ) : (
          <>
            <Link to="/login" className="btn-ghost">Sign in</Link>
            <Link to="/signup" className="btn-primary">Sign up</Link>
          </>
        )}
      </div>
    </header>
  );
}
```

CHAPTER 06 · THE FREE BUILDER COURSE

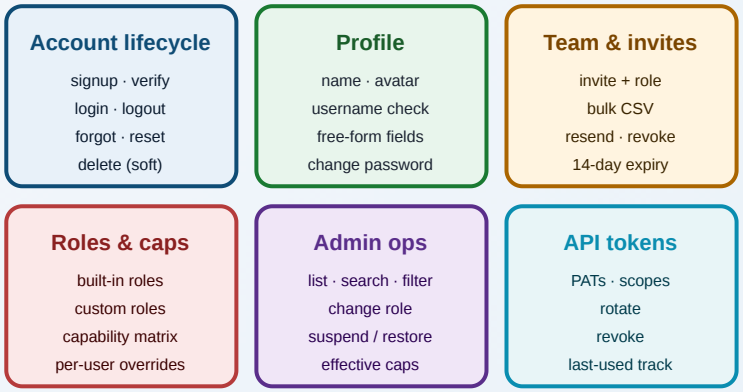
User Management, or: The Foundation You Will Stub Your Toe On

Signup, login, invites, roles, capabilities, and the one personal access token that ate Tuesday.

> *Users are not a feature. Users are the substrate everything else lives on. Get this layer wrong and every other layer wobbles. Get it right and you stop thinking about it until year three, which is what you want.*

What "user management" actually means

If you survey ten production SaaS apps, every single one has roughly the same set of user-management features. Not "should have". **Has**. They are the price of entry. If you skip any of them, you will be patching customer-support tickets about it within a month, I promise.



Six clusters. Every cluster ships in v1. The auto-showroom demo implements all of them — open the inspector and look.

🤦 **BUG FROM REAL LIFE** BidLight v0.4 shipped without password-reset email because "MVP". Within two weeks I had received 11 customer-support emails of the form "I forgot my password, please send help to my husband's email". Reader, the husband was sometimes a different husband than the one on the account. Auth flow is not optional.

The eight rules I will not let you break

- **Email + password** is the baseline. OAuth comes when you know your buyer.
- **Sessions for browser, JWT for API.** Don't crossbreed them; you will hate yourself.
- **Hash with bcrypt, cost ≥ 12.** argon2 if you're feeling fancy.
- **Verify emails.** The 30 seconds of friction saves you 30 hours of spam cleanup.
- **Rate-limit auth endpoints** behind a per-IP counter. Five attempts per minute.
- **Soft-delete users.** Hard delete is a fraud-investigation problem.
- **Personal access tokens:** hash on insert, return plaintext exactly once, prefix-only thereafter.
- **Audit every auth event.** Login, password change, role change. Always.

The capability catalog (the one trick)

Don't sprinkle `if (user.role === 'admin')` through your controllers. Build a **capability catalog** — a flat list of keys like `car.delete`, `user.suspend`, `billing.refund` — and make every gated route call `requireCapability('car.delete')`. The role-to-capability mapping lives in one file. The admin Roles page reads that catalog and renders the matrix for free.

```
// capabilities.js — single source of truth
module.exports = {
  CATALOG: [
    'car.create', 'car.update', 'car.delete', 'car.publish',
    'lead.read', 'lead.assign', 'lead.export',
    'user.invite', 'user.suspend', 'user.delete',
    'role.manage', 'billing.read', 'billing.refund',
    'analytics.read', 'admin.impersonate', /* ... */
```

```
  ],
  DEFAULTS: {
    customer: ['car.read'],
    sales:    ['car.read','car.update','lead.read','lead.assign'],
    manager:  [/* all of sales + */ 'car.delete','lead.export','user.invite'],
    admin:    ['*'],  // shorthand for "everything"
  }
};
```

▸ KEY TAKEAWAY

Build a capability catalog on Day 1. Every new feature adds a key. Every gated route checks a key. The admin UI is free. The whole permission system fits in one file and one middleware.

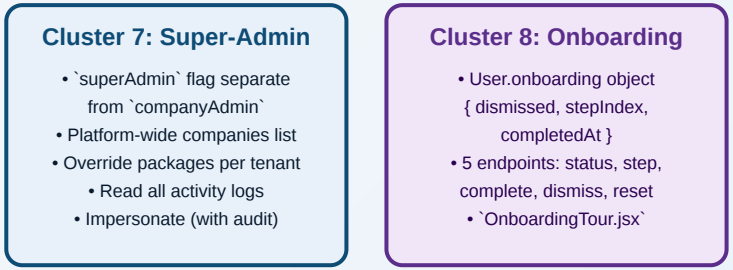
What you'll find in the demo

The Auto Showroom demo at `https://free-builder.com/cars/` ships every feature in the matrix above. Backend: `auth.js` + `users.js` + `invites.js` + `roles.js` + `access-tokens.js` — five route files, ~35 endpoints. Frontend: `Login`, `Signup`, `Profile`, `ForgotPassword`, `VerifyEmail`, `AcceptInvite`, `admin/Users`, `admin/Invites`, `admin/Roles`, `account/ApiKeys` — every page you'd expect, none you wouldn't.

Read the code files in this chapter as the implementation of every bullet above. They aren't toy — they're the same pattern that runs BidLight and Prigmar in production.

Two more clusters production apps add (and you should plan for)

The six-cluster diagram above is the v1 minimum. By month six, two more clusters earn their keep:



Two clusters BidLight skipped; Prigmar shipped both and they pay for themselves in week one.

The super-admin tier (why the `admin` flag isn't enough)

You will eventually need to do things across *all* tenants — view total revenue, override a customer's plan, debug a stuck account, ship a fix. The naive answer is "I'll just give myself `admin: true`." Don't. **Add a separate `superAdmin` flag**, and gate platform-wide routes on it.

Prigmar's `superAdmin` tier unlocks:

- `GET /superAdminPanel/companies` — list of every tenant with status + expiration + package summary.

- `PUT /superAdminPanel/companies/:id/packages` — override what a tenant can access (useful for support escalations).
- `GET /activityLogs/all` — cross-tenant audit log.
- An impersonation endpoint that logs the impersonation start/end as activity events (legal + customer-support gold).

The cost: one extra boolean on User, five admin routes, one admin page. The benefit: you don't accidentally fix a customer issue by becoming-the-customer in your own session and forgetting to log out.

Onboarding state (the feature that saves churn)

The first time a paying customer logs into your dashboard and sees seventeen empty tabs, you have roughly four seconds before they wonder if they made a mistake. Onboarding state is the fix.

Prigmar's User schema has an `onboarding` object:

```
// User.onboarding (schema fragment)
onboarding: {
  dismissed:   { type: Boolean, default: false },
  stepIndex:   { type: Number,  default: 0 },
  completedAt: { type: Date,    default: null },
}
// + 5 routes: GET/POST /onboarding/status, step, complete, dismiss, reset
```

The frontend's `OnboardingTour.jsx` reads `stepIndex`, renders the next checklist item with a tooltip pinning the relevant UI element, and POSTs back when each step completes. Five steps is enough. Ten is too many.

The API-key model (when PATs become a real product surface)

The cars-demo PAT covers the basics: one token per user, plaintext-once-on-create, hash-only on read. Prigmar takes it further because B2B customers need *programmatic* integration — Zapier, custom dashboards, internal tools.

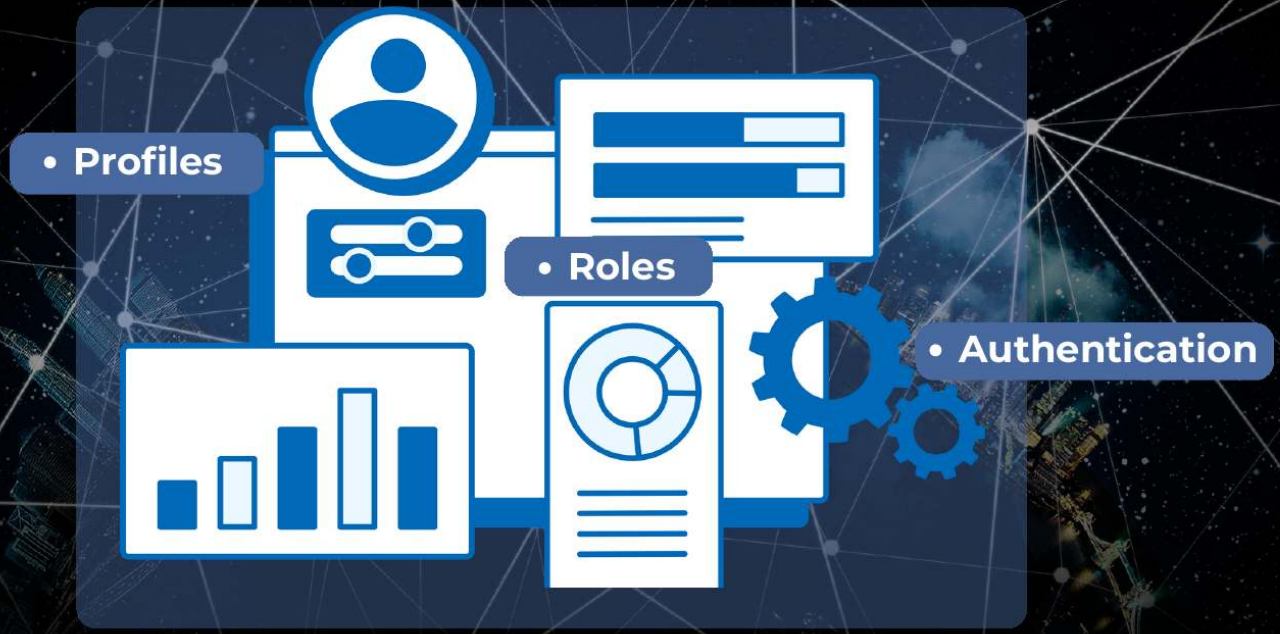
The expanded `ApiKey` model adds: `label` (so the customer can name their keys), `keyHash` (sha256), `keyPrefix` (first 8 chars, shown in the UI for identification), `scopes` array (e.g. `["contacts.read", "posts.write"]`), `lastUsedAt` + `lastUsedIp` (the user can see if a key is dead), `expiresAt` (optional auto-rotation), `revoked` (soft revoke).

Routes: `GET /apiKeys`, `POST /apiKeys` (returns plaintext exactly once), `POST /apiKeys/:id/revoke`, `DELETE /apiKeys/:id`. That's it — four endpoints, one model, and you have a real integration surface.

▸ KEY TAKEAWAY, EXPANDED

Six clusters get you to launch. The seventh (super-admin tier) keeps you sane as you scale to 50 tenants. The eighth (onboarding state) is how you stop losing first-day customers. The API-key cluster turns "we have an API" from marketing fiction into a real product surface.

USER MANAGEMENT SYSTEM



USERS ARE EVERYTHING

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE back/models/User.js

Note the `profile` field at the bottom — typed as `Mixed` with a default of `{}`. The escape hatch: new field for one experiment without a migration. Tighten the schema later when the field has earned its place. Also note the boolean flags — `admin` (tenant-scoped) is separate from `superAdmin` (platform-scoped); never collapse them.

```
// The user — extended with verification, password reset, suspension, and
// capability overrides. Mirrors the BidLight/Prigmar user model shape.
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const UserSchema = new mongoose.Schema({
  email:    { type: String, required: true, unique: true, lowercase: true, trim: true, index: true },
  username: { type: String, lowercase: true, trim: true, unique: true, sparse: true, index: true },
  password: { type: String, required: true },
  name:     { type: String },
  role:     { type: String, enum: ['customer', 'sales', 'manager', 'admin'], default: 'customer', index: true },
  roleId:   { type: mongoose.Schema.Types.ObjectId, ref: 'Role' },          // optional finer-grained role
  showroom: { type: mongoose.Schema.Types.ObjectId, ref: 'Showroom', index: true },
  profile:  { type: mongoose.Schema.Types.Mixed, default: {} },

  emailVerified:           { type: Boolean, default: false },
  emailVerificationToken:  String,

  passwordResetToken:      String,
  passwordResetExpires:    Date,

  suspended:    { type: Boolean, default: false, index: true },
  suspendedAt:  Date,
  suspendedBy:  { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  suspendedReason: String,

  extraCapabilities: { type: [String], default: [] },   // per-user grants beyond their role
  deniedCapabilities: { type: [String], default: [] },  // per-user denies (override role grants)

  invitedBy:    { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  inviteAccepted: { type: Boolean, default: true },     // false until they accept their invite

  lastLoginAt:  Date,
  lastSeenAt:   Date,
  createdAt:    { type: Date, default: Date.now },
  deletedAt:    Date,
});
```

```
UserSchema.methods.setPassword = async function (plain) {
  this.password = await bcrypt.hash(plain, 11);
  this.passwordResetToken = undefined;
  this.passwordResetExpires = undefined;
};
UserSchema.methods.checkPassword = function (plain) {
  if (!this.password) return false;
  return bcrypt.compare(plain, this.password);
};
UserSchema.methods.toSafe = function () {
  const o = this.toObject();
  delete o.password;
  delete o.emailVerificationToken;
  delete o.passwordResetToken;
  delete o.passwordResetExpires;
  return o;
};

module.exports = mongoose.model('User', UserSchema);
```

DEMO FILE back/controllers/user.ctrl.js

Auth controller. Signup, login, me, updateProfile, password change, password reset, email verify. Each returns a consistent `{ok, user, token}` envelope so the frontend never has to guess. The `activityLogs.record` calls are fire-and-forget — auth flow must never block on logging.

```
// User-account flows: signup, login, me, profile, password reset, email verify, delete account.
// Full parity with what BidLight/Prigmar expose, adapted to the car-sales domain.
const crypto = require('crypto');
const User = require('../models/User');
const Invite = require('../models/Invite');
const { signToken } = require('../utils/authentication');
const activityLogs = require('../utils/activityLogs');
const email = require('../utils/email');
const { effectiveCapabilities } = require('../utils/capabilities');

function publicOrigin(req) {
  return (req.headers && req.headers.origin) || 'http://localhost:5173';
}

exports.signup = async (req, res) => {
  const { email: e, password, name, username } = req.body || {};
  if (!e || !password) return res.status(400).json({ ok: false, error: 'Email and password required.' });
  if (password.length < 8) return res.status(400).json({ ok: false, error: 'Password must be at least 8 characters.' });
  const existing = await User.findOne({ email: String(e).toLowerCase() });
  if (existing) return res.status(409).json({ ok: false, error: 'Email already registered.' });

  const user = new User({ email: e, name, username, role: 'customer' });
  await user.setPassword(password);
  user.emailVerificationToken = crypto.randomBytes(24).toString('hex');
  await user.save();

  // Fire-and-forget verification email
  email.sendVerification({
    to: user.email,
    name: user.name,
    link: `${publicOrigin(req)}/verify-email/${user.emailVerificationToken}`,
  }).catch(() => {});

  activityLogs.record({ actor: user._id, action: 'user.signup', target: 'User', targetId: user._id, req });
  return res.status(201).json({
    ok: true,
    user: user.toSafe(),
```

```
    capabilities: effectiveCapabilities(user),
    token: signToken({ sub: user._id, role: user.role }),
  });
};

exports.login = async (req, res) => {
  const { email: e, password } = req.body || {};
  if (!e || !password) return res.status(400).json({ ok: false, error: 'Email and password required.' });
  const user = await User.findOne({ email: String(e).toLowerCase() });
  if (!user) return res.status(401).json({ ok: false, error: 'Invalid credentials.' });
  if (user.suspended) return res.status(403).json({ ok: false, error: 'Account suspended.' });
  const ok = await user.checkPassword(password);
  if (!ok) return res.status(401).json({ ok: false, error: 'Invalid credentials.' });
  user.lastLoginAt = new Date();
  user.lastSeenAt = new Date();
  await user.save();
  activityLogs.record({ actor: user._id, action: 'user.login', target: 'User', targetId: user._id, req });
  return res.json({
    ok: true,
    user: user.toSafe(),
    capabilities: effectiveCapabilities(user),
    token: signToken({ sub: user._id, role: user.role }),
  });
};

exports.logout = async (req, res) => {
  // JWT is stateless — client just drops the token. We log the event.
  if (req.user) activityLogs.record({ actor: req.user._id, action: 'user.logout', target: 'User', targetId: req.user._id, req });
  res.json({ ok: true });
};

exports.me = async (req, res) => {
  req.user.lastSeenAt = new Date();
  await req.user.save();
  res.json({ ok: true, user: req.user.toSafe(), capabilities: effectiveCapabilities(req.user) });
};

exports.updateProfile = async (req, res) => {
  const editable = ['name', 'username', 'profile'];
  for (const k of editable) if (k in req.body) req.user[k] = req.body[k];
  await req.user.save();
  activityLogs.record({ actor: req.user._id, action: 'user.profile_update', target: 'User', targetId: req.user._id, req });
  res.json({ ok: true, user: req.user.toSafe() });
};

exports.changePassword = async (req, res) => {
```

```
  const { current, next } = req.body || {};
  if (!current || !next) return res.status(400).json({ ok: false, error: 'Current and new password required.' });
  if (next.length < 8) return res.status(400).json({ ok: false, error: 'New password must be at least 8 characters.' });
  const ok = await req.user.checkPassword(current);
  if (!ok) return res.status(401).json({ ok: false, error: 'Current password is wrong.' });
  await req.user.setPassword(next);
  await req.user.save();
  activityLogs.record({ actor: req.user._id, action: 'user.password_change', target: 'User', targetId: req.user._id, req });
  res.json({ ok: true });
};

exports.requestPasswordReset = async (req, res) => {
  const e = String(req.body?.email || '').toLowerCase();
  // Always respond OK so we don't leak which emails are registered.
  if (!e) return res.json({ ok: true });
  const user = await User.findOne({ email: e });
  if (user && !user.suspended) {
    user.passwordResetToken = crypto.randomBytes(24).toString('hex');
    user.passwordResetExpires = new Date(Date.now() + 60 * 60 * 1000);  // 1 hour
    await user.save();
    email.sendPasswordReset({
      to: user.email,
      name: user.name,
      link: `${publicOrigin(req)}/reset-password/${user.passwordResetToken}`,
    }).catch(() => {});
    activityLogs.record({ actor: user._id, action: 'user.request_password_reset', target: 'User', targetId: user._id, req });
  }
  res.json({ ok: true });
};

exports.resetPassword = async (req, res) => {
  const { token, password } = req.body || {};
  if (!token || !password) return res.status(400).json({ ok: false, error: 'Token and new password required.' });
  if (password.length < 8) return res.status(400).json({ ok: false, error: 'New password must be at least 8 characters.' });
  const user = await User.findOne({
    passwordResetToken: token,
    passwordResetExpires: { $gt: new Date() },
  });
  if (!user) return res.status(400).json({ ok: false, error: 'Reset link is invalid or expired.' });
  await user.setPassword(password);
  await user.save();
  activityLogs.record({ actor: user._id, action: 'user.password_reset', target: 'User', targetId: user._id, req });
  res.json({ ok: true, token: signToken({ sub: user._id, role: user.role }), user: user.toSafe() });
};

exports.verifyEmail = async (req, res) => {
```

```
  const token = req.params.token;
  const user = await User.findOne({ emailVerificationToken: token });
  if (!user) return res.status(400).json({ ok: false, error: 'Verification link invalid.' });
  user.emailVerified = true;
  user.emailVerificationToken = undefined;
  await user.save();
  activityLogs.record({ actor: user._id, action: 'user.email_verified', target: 'User', targetId: user._id, req });
  res.json({ ok: true });
};

exports.resendVerification = async (req, res) => {
  if (req.user.emailVerified) return res.json({ ok: true, already: true });
  const crypto = require('crypto');
  req.user.emailVerificationToken = crypto.randomBytes(24).toString('hex');
  await req.user.save();
  email.sendVerification({
    to: req.user.email,
    name: req.user.name,
    link: `${publicOrigin(req)}/verify-email/${req.user.emailVerificationToken}`,
  }).catch(() => {});
  res.json({ ok: true });
};

// Verify a username is free — for live signup forms
exports.verifyUsername = async (req, res) => {
  const u = String(req.body?.username || req.query?.username || '').toLowerCase().trim();
  if (!u) return res.status(400).json({ ok: false, error: 'username required' });
  const exists = await User.findOne({ username: u });
  res.json({ ok: true, available: !exists });
};

// User deletes their own account
exports.deleteSelf = async (req, res) => {
  req.user.deletedAt = new Date();
  req.user.email = req.user.email + '.deleted.' + Date.now();   // free the email
  req.user.suspended = true;
  await req.user.save();
  activityLogs.record({ actor: req.user._id, action: 'user.delete_self', target: 'User', targetId: req.user._id, req });
  res.json({ ok: true });
};

// Admin: list users with search/filter
exports.adminList = async (req, res) => {
  const { q, role, suspended, limit = 50, skip = 0 } = req.query;
  const filter = { deletedAt: null };
  if (role) filter.role = role;
```

```
  if (suspended === 'true') filter.suspended = true;
  if (suspended === 'false') filter.suspended = false;
  if (q) filter.$or = [
    { email: new RegExp(q, 'i') },
    { name: new RegExp(q, 'i') },
    { username: new RegExp(q, 'i') },
  ];
  const items = await User.find(filter)
    .select('-password -emailVerificationToken -passwordResetToken -passwordResetExpires')
    .sort({ createdAt: -1 })
    .skip(Number(skip))
    .limit(Math.min(Number(limit), 200));
  const total = await User.countDocuments(filter);
  res.json({ ok: true, items, total });
};

// Admin: change role
exports.adminChangeRole = async (req, res) => {
  const u = await User.findById(req.params.id);
  if (!u) return res.status(404).json({ ok: false, error: 'User not found.' });
  const prev = u.role;
  u.role = req.body.role || u.role;
  await u.save();
  activityLogs.record({ actor: req.user._id, action: 'user.role_change', target: 'User', targetId: u._id, diff: { from: prev, to:
u.role }, req });
  res.json({ ok: true, user: u.toSafe() });
};

// Admin: suspend / unsuspend
exports.adminSuspend = async (req, res) => {
  const u = await User.findById(req.params.id);
  if (!u) return res.status(404).json({ ok: false, error: 'User not found.' });
  if (req.body.suspended) {
    u.suspended = true;
    u.suspendedAt = new Date();
    u.suspendedBy = req.user._id;
    u.suspendedReason = req.body.reason || null;
  } else {
    u.suspended = false;
    u.suspendedAt = null;
    u.suspendedBy = null;
    u.suspendedReason = null;
  }
  await u.save();
  activityLogs.record({ actor: req.user._id, action: u.suspended ? 'user.suspend' : 'user.unsuspend', target: 'User', targetId:
u._id, diff: { reason: u.suspendedReason }, req });
```

```
  res.json({ ok: true, user: u.toSafe() });
};

// Admin: delete a user (soft delete)
exports.adminDelete = async (req, res) => {
  const u = await User.findById(req.params.id);
  if (!u) return res.status(404).json({ ok: false, error: 'User not found.' });
  if (String(u._id) === String(req.user._id)) {
    return res.status(400).json({ ok: false, error: "Use /me/delete to delete your own account." });
  }
  u.deletedAt = new Date();
  u.suspended = true;
  u.email = u.email + '.deleted.' + Date.now();
  await u.save();
  activityLogs.record({ actor: req.user._id, action: 'user.delete', target: 'User', targetId: u._id, req });
  res.json({ ok: true });
};

// Admin: set per-user capability overrides
exports.adminSetCapabilities = async (req, res) => {
  const u = await User.findById(req.params.id);
  if (!u) return res.status(404).json({ ok: false, error: 'User not found.' });
  if (Array.isArray(req.body.extra)) u.extraCapabilities = req.body.extra;
  if (Array.isArray(req.body.denied)) u.deniedCapabilities = req.body.denied;
  await u.save();
  activityLogs.record({ actor: req.user._id, action: 'user.capabilities_set', target: 'User', targetId: u._id, diff: { extra:
u.extraCapabilities, denied: u.deniedCapabilities }, req });
  res.json({ ok: true, user: u.toSafe(), capabilities: effectiveCapabilities(u) });
};
```

DEMO FILE back/utils/capabilities.js

The capability catalog. Twenty keys, grouped by domain. The `DEFAULTS` table is the role-to-capability mapping. This file is the entire permission system; everything else is glue.

```
// The full capability catalog. Every gated action in the app maps to one key here.
// Roles reference subsets; per-user overrides extend or deny.
// Order matches the natural grouping in the admin UI.

const CAPABILITIES = [
  // -- User management --
  { key: 'user.view',     label: 'View users',        group: 'Users' },
  { key: 'user.invite',   label: 'Invite users',      group: 'Users' },
  { key: 'user.edit',     label: 'Edit user profile', group: 'Users' },
  { key: 'user.suspend',  label: 'Suspend users',     group: 'Users' },
  { key: 'user.delete',   label: 'Delete users',      group: 'Users' },
  { key: 'role.manage',   label: 'Manage roles',      group: 'Users' },

  // -- Inventory (cars) --
  { key: 'car.view',      label: 'View inventory',    group: 'Inventory' },
  { key: 'car.create',    label: 'Add cars',          group: 'Inventory' },
  { key: 'car.edit',      label: 'Edit cars',         group: 'Inventory' },
  { key: 'car.delete',    label: 'Delete cars',       group: 'Inventory' },
  { key: 'car.publish',   label: 'Publish/archive cars', group: 'Inventory' },

  // -- Leads --
  { key: 'lead.view',     label: 'View leads',        group: 'Sales' },
  { key: 'lead.update',   label: 'Update leads',      group: 'Sales' },
  { key: 'lead.assign',   label: 'Assign leads',      group: 'Sales' },
  { key: 'lead.delete',   label: 'Delete leads',      group: 'Sales' },

  // -- Billing --
  { key: 'billing.view',  label: 'View billing',      group: 'Billing' },
  { key: 'billing.manage',label: 'Manage plans',      group: 'Billing' },

  // -- Analytics --
  { key: 'analytics.view',label: 'View analytics',    group: 'Analytics' },

  // -- Showroom --
  { key: 'showroom.edit', label: 'Edit showroom',     group: 'Settings' },

  // -- API keys --
  { key: 'apikey.manage', label: 'Manage API keys',   group: 'Settings' },
```

```
];

const ALL_KEYS = CAPABILITIES.map(c => c.key);

// Default capability sets per built-in role
const ROLE_DEFAULTS = {
  customer: [],
  sales:    ['user.view', 'car.view', 'lead.view', 'lead.update', 'analytics.view'],
  manager:  [
    'user.view', 'user.invite', 'user.edit',
    'car.view', 'car.create', 'car.edit', 'car.publish',
    'lead.view', 'lead.update', 'lead.assign', 'lead.delete',
    'analytics.view',
  ],
  admin:    ALL_KEYS,
};

function effectiveCapabilities(user) {
  if (!user) return [];
  if (user.role === 'admin') return ALL_KEYS;
  const base = new Set(ROLE_DEFAULTS[user.role] || []);
  (user.extraCapabilities || []).forEach(c => base.add(c));
  (user.deniedCapabilities || []).forEach(c => base.delete(c));
  return Array.from(base);
}

function hasCapability(user, key) {
  if (!user) return false;
  if (user.role === 'admin') return true;
  const eff = effectiveCapabilities(user);
  return eff.includes(key);
}

module.exports = { CAPABILITIES, ALL_KEYS, ROLE_DEFAULTS, effectiveCapabilities, hasCapability };
```

DEMO FILE back/controllers/invite.ctrl.js

Invite flow. Create single, create bulk (CSV-style array), list, accept, decline, resend, revoke. The 14-day expiry is enforced on accept, not on cron — fewer moving parts.

```
// Invite flows: create, list, accept, decline, revoke, resend, bulk CSV import.
// Same shape as BidLight/Prigmar invites.
const crypto = require('crypto');
const Invite = require('../models/Invite');
const Role = require('../models/Role');
const User = require('../models/User');
const { signToken } = require('../utils/authentication');
const activityLogs = require('../utils/activityLogs');
const email = require('../utils/email');

const INVITE_TTL_DAYS = 14;

function newToken() { return crypto.randomBytes(24).toString('hex'); }

exports.create = async (req, res) => {
  const { email: e, name, roleSlug, roleId, message } = req.body || {};
  if (!e) return res.status(400).json({ ok: false, error: 'Email is required.' });

  // Don't re-invite an existing user with the same email
  const existing = await User.findOne({ email: String(e).toLowerCase() });
  if (existing) return res.status(409).json({ ok: false, error: 'User with this email already exists.' });

  let role = null;
  if (roleId) role = await Role.findById(roleId);
  else if (roleSlug) role = await Role.findOne({ slug: roleSlug });

  const invite = await Invite.create({
    token:    newToken(),
    email:    String(e).toLowerCase(),
    name:     name || null,
    roleId:   role ? role._id : null,
    roleSlug: role ? role.slug : (roleSlug || 'sales'),
    showroom: req.user.showroom || null,
    invitedBy: req.user._id,
    message:  message || null,
    expiresAt: new Date(Date.now() + INVITE_TTL_DAYS * 24 * 60 * 60 * 1000),
  });
```

```
  email.sendInvite({
    to: invite.email,
    name: invite.name,
    inviterName: req.user.name || req.user.email,
    role: invite.roleSlug,
    link: `${(req.headers.origin || 'http://localhost:5173')}/accept-invite/${invite.token}`,
    message: invite.message,
  }).catch(() => {});

  activityLogs.record({ actor: req.user._id, action: 'invite.create', target: 'Invite', targetId: invite._id, diff: { email:
invite.email, role: invite.roleSlug }, req });
  res.status(201).json({ ok: true, invite });
};

exports.list = async (req, res) => {
  const { status, q, limit = 50, skip = 0 } = req.query;
  const filter = {};
  if (q) filter.$or = [{ email: new RegExp(q, 'i') }, { name: new RegExp(q, 'i') }];

  let items = await Invite.find(filter)
    .sort({ createdAt: -1 })
    .skip(Number(skip)).limit(Math.min(Number(limit), 200))
    .populate('invitedBy', 'email name')
    .populate('roleId', 'name slug');
  // Filter by status virtual after fetch
  if (status) items = items.filter(i => i.status === status);
  res.json({ ok: true, items, total: items.length });
};

exports.getByToken = async (req, res) => {
  const inv = await Invite.findOne({ token: req.params.token })
    .populate('invitedBy', 'email name')
    .populate('roleId', 'name slug');
  if (!inv) return res.status(404).json({ ok: false, error: 'Invite not found.' });
  res.json({ ok: true, invite: inv });
};

exports.accept = async (req, res) => {
  const { token, password, name, username } = req.body || {};
  const inv = await Invite.findOne({ token });
  if (!inv) return res.status(404).json({ ok: false, error: 'Invite not found.' });
  if (inv.status !== 'pending') return res.status(400).json({ ok: false, error: `Invite is ${inv.status}.` });

  if (!password || password.length < 8) return res.status(400).json({ ok: false, error: 'Password must be at least 8 characters.'
});
```

```
  let user = await User.findOne({ email: inv.email });
  if (user) return res.status(409).json({ ok: false, error: 'User with this email already exists.' });

  user = new User({
    email: inv.email,
    name: name || inv.name,
    username,
    role: inv.roleSlug || 'sales',
    showroom: inv.showroom,
    invitedBy: inv.invitedBy,
    inviteAccepted: true,
    emailVerified: true,                  // they came from the invite email
  });
  await user.setPassword(password);
  await user.save();

  inv.acceptedAt = new Date();
  inv.acceptedBy = user._id;
  await inv.save();

  activityLogs.record({ actor: user._id, action: 'invite.accept', target: 'Invite', targetId: inv._id, req });
  res.status(201).json({ ok: true, user: user.toSafe(), token: signToken({ sub: user._id, role: user.role }) });
};

exports.decline = async (req, res) => {
  const inv = await Invite.findOne({ token: req.params.token });
  if (!inv) return res.status(404).json({ ok: false, error: 'Invite not found.' });
  inv.declinedAt = new Date();
  await inv.save();
  res.json({ ok: true });
};

exports.revoke = async (req, res) => {
  const inv = await Invite.findById(req.params.id);
  if (!inv) return res.status(404).json({ ok: false, error: 'Invite not found.' });
  inv.revokedAt = new Date();
  inv.revokedBy = req.user._id;
  await inv.save();
  activityLogs.record({ actor: req.user._id, action: 'invite.revoke', target: 'Invite', targetId: inv._id, req });
  res.json({ ok: true });
};

exports.resend = async (req, res) => {
  const inv = await Invite.findById(req.params.id);
  if (!inv) return res.status(404).json({ ok: false, error: 'Invite not found.' });
  if (inv.status !== 'pending') return res.status(400).json({ ok: false, error: `Invite is ${inv.status}.` });
```

```
  inv.remindersSent = (inv.remindersSent || 0) + 1;
  inv.expiresAt = new Date(Date.now() + INVITE_TTL_DAYS * 24 * 60 * 60 * 1000);
  await inv.save();
  email.sendInvite({
    to: inv.email,
    name: inv.name,
    inviterName: req.user.name || req.user.email,
    role: inv.roleSlug,
    link: `${(req.headers.origin || 'http://localhost:5173')}/accept-invite/${inv.token}`,
    message: inv.message,
  }).catch(() => {});
  activityLogs.record({ actor: req.user._id, action: 'invite.resend', target: 'Invite', targetId: inv._id, req });
  res.json({ ok: true, invite: inv });
};

// Bulk: accept a list of {email, name?, roleSlug?} rows and create invites
exports.bulkCreate = async (req, res) => {
  const rows = Array.isArray(req.body?.rows) ? req.body.rows : [];
  if (rows.length === 0) return res.status(400).json({ ok: false, error: 'No rows provided.' });
  if (rows.length > 200) return res.status(400).json({ ok: false, error: 'Too many rows (max 200).' });

  const created = [], skipped = [];
  for (const row of rows) {
    const e = String(row.email || '').toLowerCase().trim();
    if (!e || !/.+@.+\..+/.test(e)) { skipped.push({ row, reason: 'bad email' }); continue; }
    if (await User.findOne({ email: e })) { skipped.push({ row, reason: 'user exists' }); continue; }
    if (await Invite.findOne({ email: e, acceptedAt: null, revokedAt: null, declinedAt: null, expiresAt: { $gt: new Date() } }))
{
      skipped.push({ row, reason: 'invite already pending' }); continue;
    }
    const inv = await Invite.create({
      token: newToken(), email: e, name: row.name || null,
      roleSlug: row.roleSlug || 'sales',
      showroom: req.user.showroom || null,
      invitedBy: req.user._id,
      source: 'csv',
      expiresAt: new Date(Date.now() + INVITE_TTL_DAYS * 24 * 60 * 60 * 1000),
    });
    created.push(inv);
  }
  activityLogs.record({ actor: req.user._id, action: 'invite.bulk_create', target: 'Invite', diff: { count: created.length,
skipped: skipped.length }, req });
  res.status(201).json({ ok: true, created, skipped });
};
```

DEMO FILE back/controllers/accessToken.ctrl.js

PATs in 80 lines. **Plaintext returned exactly once** on create — after that, only the `prefix` is shown in the UI. The hashed token is what we compare on auth. This is how GitHub does it; this is how you should do it. Prigmar extends this with `scopes`, `lastUsedAt`, `expiresAt`, and `revoked` for production-grade integration.

```
// Personal access tokens. Like GitHub PATs.
// We return the full plaintext exactly once at creation time; after that, only the prefix.
const crypto = require('crypto');
const AccessToken = require('../models/AccessToken');
const activityLogs = require('../utils/activityLogs');

function sha256(s) { return crypto.createHash('sha256').update(s).digest('hex'); }
function newPlaintextToken() {
  // ats_ prefix + 32 hex = 36 chars total
  return 'ats_' + crypto.randomBytes(16).toString('hex');
}

exports.list = async (req, res) => {
  const items = await AccessToken.find({ user: req.user._id, revoked: false })
    .select('-tokenHash')
    .sort({ createdAt: -1 });
  res.json({ ok: true, items });
};

exports.create = async (req, res) => {
  const { name, scopes, expiresAt } = req.body || {};
  if (!name) return res.status(400).json({ ok: false, error: 'name required' });
  const plain = newPlaintextToken();
  const t = await AccessToken.create({
    user: req.user._id,
    showroom: req.user.showroom || null,
    name,
    prefix: plain.slice(0, 12),
    tokenHash: sha256(plain),
    scopes: Array.isArray(scopes) ? scopes : ['read'],
    expiresAt: expiresAt ? new Date(expiresAt) : null,
  });
  activityLogs.record({ actor: req.user._id, action: 'apikey.create', target: 'AccessToken', targetId: t._id, diff: { name,
scopes: t.scopes }, req });
  // Return plaintext ONCE
  const safe = t.toObject();
  delete safe.tokenHash;
```

```
  safe.token = plain;
  res.status(201).json({ ok: true, token: safe });
};

exports.revoke = async (req, res) => {
  const t = await AccessToken.findOne({ _id: req.params.id, user: req.user._id });
  if (!t) return res.status(404).json({ ok: false, error: 'Token not found.' });
  t.revoked = true;
  t.revokedAt = new Date();
  t.revokedBy = req.user._id;
  await t.save();
  activityLogs.record({ actor: req.user._id, action: 'apikey.revoke', target: 'AccessToken', targetId: t._id, req });
  res.json({ ok: true });
};

exports.rotate = async (req, res) => {
  const old = await AccessToken.findOne({ _id: req.params.id, user: req.user._id });
  if (!old) return res.status(404).json({ ok: false, error: 'Token not found.' });
  const plain = newPlaintextToken();
  old.prefix = plain.slice(0, 12);
  old.tokenHash = sha256(plain);
  await old.save();
  activityLogs.record({ actor: req.user._id, action: 'apikey.rotate', target: 'AccessToken', targetId: old._id, req });
  const safe = old.toObject();
  delete safe.tokenHash;
  safe.token = plain;
  res.json({ ok: true, token: safe });
};
```

CHAPTER 07 · THE FREE BUILDER COURSE

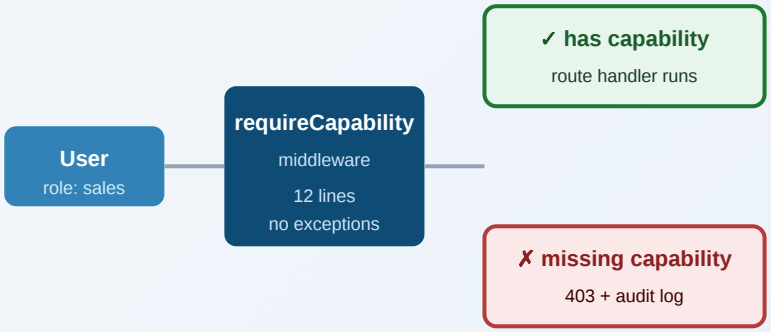
Permission Management, or: How To Not Email A Customer's Data To The Wrong Customer

Roles + capabilities + middleware, in that order. Anything else is a future Monday morning.

> *There are two kinds of permission bugs. The kind where a logged-in user can't do something they should be able to — annoying, fixable, gets caught in QA. And the kind where a logged-in user can do something they shouldn't — career-defining, lawyer-involving, gets caught by your worst customer.*

The framework, in one sentence

One role per user. A capability catalog mapping fine-grained actions to keys. Middleware that checks the key on every gated route. That's it. Anything else is over-engineering before you have a permission problem.



One in, two out. The middleware is the entire permission system. Everything upstream is data; everything downstream is glue.

The five anti-patterns I will judge you for

- **Boolean fields on User** (`isAdmin` , `canExport`) — you'll be migrating these for a decade.
- **Scattered** `if (role==='admin')` — Ctrl+F nightmare in year two.
- **Frontend permission checks only** — your client is not your friend.
- **Permission strings as raw text** (`"can edit cars"`) — typos = silent privilege escalation.
- **One mega-role per user** with no override mechanism — you'll create a sub-role for the one exception and it will never end.

> 😬 **WAR STORY** A previous version of Prigmar had a frontend-only check: `if (user.role === 'admin') showDeleteButton()` . The button was hidden, but the `DELETE` endpoint had no server check. A power user noticed the button missing, opened DevTools, called the endpoint directly, and deleted their entire workspace by accident. Three hours to restore from backup. Server-side checks are not optional.

Roles vs capabilities — the difference matters

A **role** is a label. `sales`, `manager`, `admin`. Useful for human conversation: "Joe is a manager." A **capability** is the actual atomic permission. `car.delete`, `user.suspend`, `billing.refund`. Useful for the machine: "Can Joe do `car.delete`? Yes / no."

Roles bundle capabilities. The mapping lives in *one file*. When sales should suddenly be able to export leads, you add `lead.export` to the `sales` bundle — not to every controller. This is the single biggest architectural win in the whole user-management layer.

Plan-tier gating uses the same gun

When you're on the Starter plan and a feature is Pro-only, that is also a permission check. Don't build a second system for it. Add a capability key like `analytics.advanced`, attach it to plans the same way you attach it to roles, and use the same middleware. One gun, two triggers.

Three plans. Same capability catalog. New capability = new feature; new plan = new bundle.

The audit log is the safety net

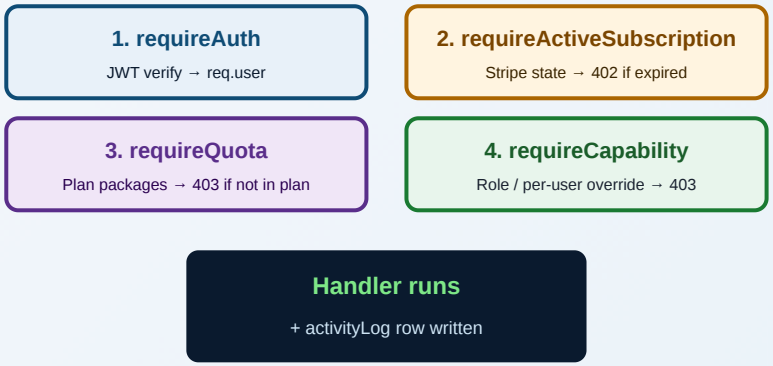
Every gated action — successful and failed — should write one row to your activity log. `actor`, `action`, `target`, `diff`, `ip`. When (not if) someone says "I didn't do that", you go to the log. When (not if) someone tries to escalate privileges, you go to the log. The log is cheap; lawsuits are not.

> ▸ KEY TAKEAWAY
>
> Roles for humans, capabilities for the machine, middleware for the enforcement, audit log for the cleanup. Build it in this order and the permission system disappears from your daily worries.

The four-middleware stack (a production pattern, layered)

One middleware is enough for a hobby project. A shipping SaaS layers four — each handling a different concern — composed into a route's signature like this:

```
router.delete('/cars/:id',
  requireAuth(),                  // 1. Who are you?
  requireActiveSubscription(),    // 2. Have you paid this month?
  requireQuota('car.delete'),     // 3. Does your plan allow this action?
  requireCapability('car.delete'),// 4. Does your role allow it within your plan?
  controllers.cars.delete);
```



Four checks. Each answers a different question. Each can short-circuit with the right status code (401 / 402 / 403 / 403).

What each middleware actually does

- **`requireAuth`** (12 lines) — JWT verify, user lookup, attach to `req.user`, `next()`. Returns 401 if the token is missing or expired.
- **`requireActiveSubscription`** (15 lines) — looks up the user's company, checks `expiration > now`. Returns 402 Payment Required with a JSON envelope the frontend renders as the "your subscription expired" modal. Skipped for super-admins (they pay differently).
- **`requireQuota(capability)`** (20 lines) — checks whether the user's *plan packages* include the requested capability. This is plan-tier gating: Starter doesn't include `analytics.advanced`, regardless of role. Returns 403 with a "upgrade required" envelope that includes the relevant plan's upgrade URL.
- **`requireCapability(capability)`** (10 lines) — checks whether the user's *role* (or per-user override) includes the capability. This is role-based gating within a plan. Returns 403 with an "ask your admin" message.

Note the order: auth → subscription → plan → role. Each layer assumes the previous one passed. If you reorder them, the error messages stop making sense (a 403 "your role doesn't allow this" when the user actually needs to pay is the worst customer-support ticket I've ever generated).

The 17 endpoints behind the matrix

The cars-demo Roles page is the visible tip. Underneath it sit two route files implementing the data plane:

- **`/api/permissions/*`** (13 endpoints): get by entityId (where entityId is a user, role, or company), get full catalog, fetch user+company merged set, fetch admin-only set, create, create-default, add per-user override, upsert for entity, copy role-permissions to a user, toggle a single permission, soft-delete.
- **`/api/roles/*`** (4 endpoints): list company roles (built-ins merged with custom), get one, create custom role, update role for a specific user.

That's 17 endpoints implementing what reads as one page in the UI. Permission systems are an iceberg; the visible part is small.

The API-key gate (one more middleware)

If you ship a public API — even just for Zapier — you need a fifth middleware variant: `requireApiKey()`. Same shape as `requireAuth`, but it reads the `Authorization: Bearer ak_xxx` header, hashes the token, looks up the matching `ApiKey` document, verifies it's not revoked and not expired, attaches the owning user + the key's scopes to `req`, and updates `lastUsedAt` + `lastUsedIp` fire-and-forget.

The trick: API-key routes use `requireApiKey()` INSTEAD OF `requireAuth()`, then layer the same `requireActiveSubscription` + `requireQuota` + `requireCapability` on top. Same gates, different identity source. One framework, two doors.

▸ KEY TAKEAWAY, EXPANDED

Four middlewares, in this order: auth → subscription → plan → role. Each can short-circuit with its own status code, each has its own audit-log signal. Add `requireApiKey` as the fifth when you open up programmatic access. The whole permission system is five middlewares and one capability catalog. That's it.

PERMISSION MANAGEMENT

Control Access, Enable Features

Control what people can do inside your platform

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE `back/middlewares/requireAuth.js`

Twelve lines. JWT verify → user lookup → attach to `req.user` → `next()`. Auth middleware should be boring. If yours is complicated, it's hiding a bug.

```
// Verifies a JWT, loads the user, and attaches { user } to the request.
// Reject early with 401 if missing/invalid.
const User = require('../models/User');
const { verifyToken, tokenFromRequest } = require('../utils/authentication');

module.exports = async function requireAuth(req, res, next) {
  const token = tokenFromRequest(req);
  if (!token) return res.status(401).json({ ok: false, error: 'Sign in required.' });
  const payload = verifyToken(token);
  if (!payload || !payload.sub) return res.status(401).json({ ok: false, error: 'Invalid session.' });
  const user = await User.findById(payload.sub);
  if (!user) return res.status(401).json({ ok: false, error: 'User not found.' });
  req.user = user;
  next();
};
```

DEMO FILE back/middlewares/requirePermission.js

Accepts a role string, an array of roles, or a custom predicate function. The function form is the escape hatch: when you need "the user is allowed if they own this resource", you pass a closure. Admins always pass — keep that consistent or you'll lock yourself out at 3 AM.

```
// Role + capability gate. Use after requireAuth.
//   requirePermission('admin')                 → must be admin
//   requirePermission(['manager', 'admin'])    → any of these roles
//   requirePermission((user, req) => boolean)  → custom predicate
module.exports = function requirePermission(rule) {
  return (req, res, next) => {
    if (!req.user) return res.status(401).json({ ok: false, error: 'Sign in required.' });
    let allowed = false;
    if (typeof rule === 'string') allowed = req.user.role === rule || req.user.role === 'admin';
    else if (Array.isArray(rule)) allowed = rule.includes(req.user.role) || req.user.role === 'admin';
    else if (typeof rule === 'function') allowed = !!rule(req.user, req);
    if (!allowed) return res.status(403).json({ ok: false, error: 'Forbidden.' });
    next();
  };
};
```

DEMO FILE back/middlewares/requireCapability.js

The capability-aware variant. Loads the user's role, expands to the capability set, checks the requested key. The trick: it falls back to the role's default capability set *unless* a per-user override exists. Custom roles override defaults; per-user overrides override the role. All three layers, ten lines.

```
// Capability-level gate. Use after requireAuth.
//   requireCapability('user.delete')           — must have this capability
//   requireCapability(['car.edit','car.publish']) — needs all listed
const { hasCapability } = require('../utils/capabilities');

module.exports = function requireCapability(keyOrKeys) {
  const keys = Array.isArray(keyOrKeys) ? keyOrKeys : [keyOrKeys];
  return (req, res, next) => {
    if (!req.user) return res.status(401).json({ ok: false, error: 'Sign in required.' });
    if (req.user.suspended) return res.status(403).json({ ok: false, error: 'Account suspended.' });
    for (const k of keys) {
      if (!hasCapability(req.user, k)) {
        return res.status(403).json({ ok: false, error: `Missing capability: ${k}` });
      }
    }
    next();
  };
};
```

DEMO FILE back/utils/capabilities.js

Single source of truth: the capability catalog and the role-to-capability defaults. Every new gated feature adds one line here. The frontend Roles page reads this file's exports to render the permission matrix automatically. Prigmar's production version expands this with per-plan capability bundles — the same file, two layers (role bundle + plan bundle).

```
// The full capability catalog. Every gated action in the app maps to one key here.
// Roles reference subsets; per-user overrides extend or deny.
// Order matches the natural grouping in the admin UI.

const CAPABILITIES = [
  // -- User management --
  { key: 'user.view',     label: 'View users',        group: 'Users' },
  { key: 'user.invite',   label: 'Invite users',      group: 'Users' },
  { key: 'user.edit',     label: 'Edit user profile', group: 'Users' },
  { key: 'user.suspend',  label: 'Suspend users',     group: 'Users' },
  { key: 'user.delete',   label: 'Delete users',      group: 'Users' },
  { key: 'role.manage',   label: 'Manage roles',      group: 'Users' },

  // -- Inventory (cars) --
  { key: 'car.view',      label: 'View inventory',    group: 'Inventory' },
  { key: 'car.create',    label: 'Add cars',          group: 'Inventory' },
  { key: 'car.edit',      label: 'Edit cars',         group: 'Inventory' },
  { key: 'car.delete',    label: 'Delete cars',       group: 'Inventory' },
  { key: 'car.publish',   label: 'Publish/archive cars', group: 'Inventory' },

  // -- Leads --
  { key: 'lead.view',     label: 'View leads',        group: 'Sales' },
  { key: 'lead.update',   label: 'Update leads',      group: 'Sales' },
  { key: 'lead.assign',   label: 'Assign leads',      group: 'Sales' },
  { key: 'lead.delete',   label: 'Delete leads',      group: 'Sales' },

  // -- Billing --
  { key: 'billing.view',  label: 'View billing',      group: 'Billing' },
  { key: 'billing.manage',label: 'Manage plans',      group: 'Billing' },

  // -- Analytics --
  { key: 'analytics.view',label: 'View analytics',    group: 'Analytics' },

  // -- Showroom --
  { key: 'showroom.edit', label: 'Edit showroom',     group: 'Settings' },

  // -- API keys --
```

```
  { key: 'apikey.manage', label: 'Manage API keys',   group: 'Settings' },
];

const ALL_KEYS = CAPABILITIES.map(c => c.key);

// Default capability sets per built-in role
const ROLE_DEFAULTS = {
  customer: [],
  sales:    ['user.view', 'car.view', 'lead.view', 'lead.update', 'analytics.view'],
  manager:  [
    'user.view', 'user.invite', 'user.edit',
    'car.view', 'car.create', 'car.edit', 'car.publish',
    'lead.view', 'lead.update', 'lead.assign', 'lead.delete',
    'analytics.view',
  ],
  admin:    ALL_KEYS,
};

function effectiveCapabilities(user) {
  if (!user) return [];
  if (user.role === 'admin') return ALL_KEYS;
  const base = new Set(ROLE_DEFAULTS[user.role] || []);
  (user.extraCapabilities || []).forEach(c => base.add(c));
  (user.deniedCapabilities || []).forEach(c => base.delete(c));
  return Array.from(base);
}

function hasCapability(user, key) {
  if (!user) return false;
  if (user.role === 'admin') return true;
  const eff = effectiveCapabilities(user);
  return eff.includes(key);
}

module.exports = { CAPABILITIES, ALL_KEYS, ROLE_DEFAULTS, effectiveCapabilities, hasCapability };
```

CHAPTER 08 · THE FREE BUILDER COURSE

Project Management & Admin Tools — Your Cockpit, Not A Feature

The CRUD layer your customers use, plus the operator console you live in.

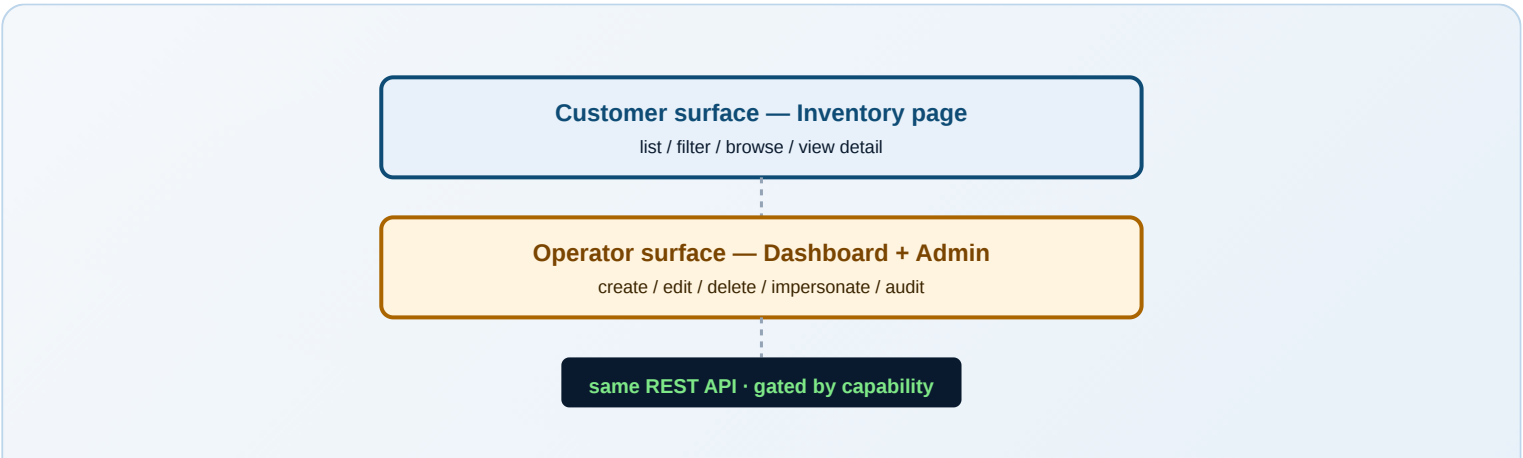
> *Every SaaS has a "project". The thing customers create, fill, share, and pay for. In BidLight it's a construction project. In Prigmar it's a marketing brand. In the auto showroom demo it's a car listing. Same shape, different costume.*

The CRUD that powers the product

Every operator builds the same five endpoints on the core object, in the same order, and either calls them by their HTTP verbs or pretends they invented something cleverer:

- `GET /api/cars` — list, with search/filter/pagination
- `GET /api/cars/:id` — read one, with related data
- `POST /api/cars` — create, returns the new id
- `PUT /api/cars/:id` — update, returns the new state
- `DELETE /api/cars/:id` — soft-delete (status='archived'), not actual delete

Boring. Predictable. Implemented in 80 lines. The boring part is the part that ships.



Two surfaces on the same data. The capability check is the wall between them.

The admin layer (your sanity)

Here's the part most tutorials skip: **you also need an interface for yourself.** Not Mongo Compass, not `ssh prod && mongo`. A real admin page where you can:

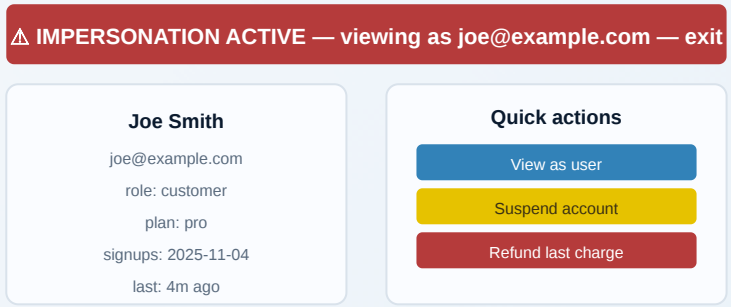
- See every user's recent activity without opening the database
- Suspend a user when they're abusing the system
- Refund a payment without logging into Stripe
- Impersonate a customer to reproduce their bug
- Seed a fresh demo inventory when your sales call is in 11 minutes

> ⏱ **TIME MATH** I once built a "support feature" — viewing a user's last 20 activity log entries on one page. Took 40 minutes. Saved me ~5 hours/week of "what did the user actually click?" debugging. Payback period: one week. Why is this not the first thing every solo SaaS builds? I don't know. I am asking everyone, including you.

The super-admin layer (handle with both hands)

Super-admin = impersonation mode + dangerous operations + raw data access. The 1% of your features that can ruin everything. Three rules:

- **Every super-admin action lands in the audit log.** Including read-only ones. Especially read-only ones.
- **Impersonation has a giant banner.** "You are viewing as Joe Smith." Red, sticky, top of screen. If you forget, you'll send Joe an email from his own account by accident.
- **Dangerous ops require confirmation.** "Type DELETE to delete." Three seconds of friction, thirty hours of disasters avoided.



The operator's cockpit. Profile on the left, dangerous buttons on the right, big red banner on top whenever you're inside someone else's account.

Seed your demo, every time

One small thing that punches above its weight: an idempotent `POST /api/admin/seed-demo` endpoint. It checks `Car.countDocuments() === 0` and, if true, creates 20 plausible cars. If false, it does nothing. Mount it on a button in the admin page labeled "Seed demo data".

Now: every fresh environment, every demo call, every screenshot you take for marketing — one click and you have data that looks like a real business. Cost: 30 minutes to write. Time saved across the next two years: I cannot even estimate.

▸ KEY TAKEAWAY

The CRUD layer is for customers; the admin layer is for you. Both ship in v1. Skipping the admin layer is how solo SaaS founders end up SSH-ing into prod at midnight. Don't be that founder. Build the cockpit.

The enriched activity log (what BidLight schema-evolved into)

The cars-demo `ActivityLog` model has the basics: `actor`, `action`, `target`, `timestamp`. Fine for v1. By the time BidLight grew up, the schema had grown to ten fields, each earning its place after a specific incident:

```
// ActivityLog (Prigmar production schema)
{
  actor:          ObjectId,   // who did it
  action:         String,     // 'car.delete', 'user.suspend', etc.
  target:         String,     // resource the action operated on
  brandId:        ObjectId,   // tenant scope (multi-tenant lookups)
  metadata:       Mixed,      // free-form context: old/new values
  isNotification: Boolean,    // should this appear in the user's bell?
  method:         String,     // HTTP method
  path:           String,     // request URL
  status:         Number,     // HTTP response code
  durationMs:     Number,     // perf: where's the slow request?
  idempotencyKey: String,     // dedupe across retries
  createdAt:      Date,
}
```

Every field tells a story. `brandId` exists because cross-tenant queries got slow without an index. `isNotification` exists because the same row can be both an audit entry AND a "you have a new follower" bell notification — one schema, two readers. `idempotencyKey` exists because Stripe-style retry-safe operations need to know "we already saw this request."

The filter views feature (saved searches earn their keep)

Once your admin table has more than 50 rows, customers start saving filter combos: "Active sales reps in Region 3 with leads > 5." Build it once, every customer reuses it. Don't build it, every customer asks for it.

The implementation: a `FilterView` collection storing `{ owner, scope (company/project/timeline), name, query (the filter DSL), columns, sort, isDefault }`. Four endpoints: list, get-by-scope, create, delete. The frontend reads "saved views" into a dropdown and applies them to the table. Twenty hours of work, lifetime customer-support savings.

Alert config (the channel that closes the loop)

Your activity log records what happened. Customers want to *be told* when specific things happen. Build a tiny alert-config layer:

- `GET /alertConfig` — read the current settings (per company, per user)
- `PUT /alertConfig` — update which events go to email, in-app, both, or neither
- `POST /alertConfig/test` — send a fake alert to verify the channel works (every email setup screen needs this button)

Now the activity log is dual-purpose: a writer (every controller writes rows) and a reader (the alert engine reads new rows, matches against config, dispatches). One table, two consumers.

Document-control / model-health (when the domain demands it)

The framework is generic, but real verticals add domain-specific admin tools. BidLight (construction) ships:

- **Document Control** — track sheet drawings: `sheetName, sheetNumber, revisionNumber, discipline, package, sheetStatus, link, isPlaceholder, floorLevel, projectId, companyId, deleted`. Seven endpoints (CRUD + bulk + filter+search+paginate+sort).
- **Discipline lookup** — master data for the trade categories (Architectural, Structural, Mechanical, etc.) with CSV upload + bulk delete.
- **Model Health** — settings (~48KB schema describing rules) + reports (running those rules against an uploaded BIM model and storing the results).

You won't ship these if you're not BidLight. But the pattern repeats in every vertical: a couple of small lookup tables, a CRUD with bulk + CSV, and a "run the rules" engine. Build them when your customers ask twice; not before.

▸ KEY TAKEAWAY, EXPANDED

Customer CRUD is the visible part. The invisible parts — super-admin tier, enriched activity log, filter views, alert config, domain admin tools — are what turn a working app into a business that runs without you on weekends.

PROJECT MANAGEMENT FEATURES

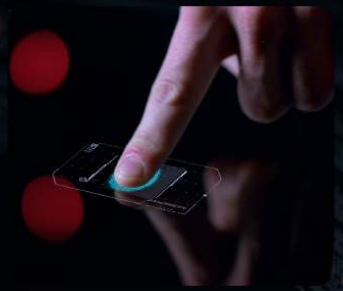
Organize the Work

Project management tools focus on tasks, subtasks, and timelines.

CRM platforms manage contacts, leads, and pipelines.

File-sharing services organize documents and media files.

ADMIN + SUPER ADMIN ACCESS

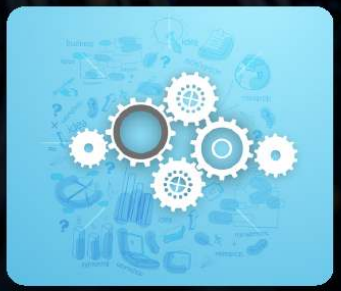


user signups

Permissions and Access Control

System Logs and Activity Feed

Analytics and Performance Metrics



Feature Toggles and Settings

Security Controls

As a business owner, you need visibility and control

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE back/controllers/car.ctrl.js

Five functions: list, get, create, update, remove. `list` is the one that gets complicated first — note the filter shape, then the `$or` regex for search. `get` increments the view counter with a fire-and-forget update (we don't await it because the response should not wait on analytics).

```
// CRUD for the core object. Public list/get; create/update/delete are staff-only.
const Car = require('../models/Car');
const activityLogs = require('../utils/activityLogs');

exports.list = async (req, res) => {
  const { status, make, bodyStyle, minPrice, maxPrice, q, limit = 24, skip = 0 } = req.query;
  const filter = {};
  if (status) filter.status = status; else filter.status = 'available';
  if (make) filter.make = make;
  if (bodyStyle) filter.bodyStyle = bodyStyle;
  if (minPrice || maxPrice) {
    filter.price = {};
    if (minPrice) filter.price.$gte = Number(minPrice);
    if (maxPrice) filter.price.$lte = Number(maxPrice);
  }
  if (q) filter.$or = [
    { make: new RegExp(q, 'i') }, { model: new RegExp(q, 'i') }, { description: new RegExp(q, 'i') }
  ];
  const items = await Car.find(filter).sort({ createdAt: -1 }).skip(Number(skip)).limit(Math.min(Number(limit), 100));
  const total = await Car.countDocuments(filter);
  res.json({ ok: true, items, total });
};

exports.get = async (req, res) => {
  const car = await Car.findById(req.params.id);
  if (!car) return res.status(404).json({ ok: false, error: 'Not found.' });
  Car.updateOne({ _id: car._id }, { $inc: { views: 1 } }).catch(() => {});
  res.json({ ok: true, car });
};

exports.create = async (req, res) => {
  const car = await Car.create({ ...req.body, createdBy: req.user._id });
  activityLogs.record({ actor: req.user._id, action: 'car.create', target: 'Car', targetId: car._id, req });
  res.status(201).json({ ok: true, car });
};

exports.update = async (req, res) => {
  const car = await Car.findById(req.params.id);
```

```
  if (!car) return res.status(404).json({ ok: false, error: 'Not found.' });
  Object.assign(car, req.body);
  await car.save();
  activityLogs.record({ actor: req.user._id, action: 'car.update', target: 'Car', targetId: car._id, diff: req.body, req });
  res.json({ ok: true, car });
};

exports.remove = async (req, res) => {
  await Car.deleteOne({ _id: req.params.id });
  activityLogs.record({ actor: req.user._id, action: 'car.delete', target: 'Car', targetId: req.params.id, req });
  res.json({ ok: true });
};
```

DEMO FILE back/routes/adminPanel.js

Two routes: `/system` (env probe) and `/seed-demo` (idempotent bootstrap). The seed function is the kind of thing that lives at the boundary between "dev convenience" and "production tool" — it's gated by `requirePermission('admin')` AND checks `countDocuments > 0` before seeding, so it can never overwrite real data.

```
// Admin-only convenience endpoints: seed demo data, impersonate, system info.
const router = require('express').Router();
const requireAuth = require('../middlewares/requireAuth');
const requirePermission = require('../middlewares/requirePermission');
const User = require('../models/User');
const Car = require('../models/Car');
const Showroom = require('../models/Showroom');

router.use(requireAuth, requirePermission('admin'));

router.get('/system', (req, res) => {
  res.json({
    ok: true,
    uptime: process.uptime(),
    node: process.version,
    memory: process.memoryUsage(),
  });
});

router.post('/seed-demo', async (req, res) => {
  // Idempotent — only seeds if empty
  const count = await Car.countDocuments({});
  if (count > 0) return res.json({ ok: true, seeded: false, reason: 'inventory not empty' });

  const showroom = await Showroom.findOneAndUpdate(
    { slug: 'main' },
    { slug: 'main', name: 'Auto Showroom — Main Floor', plan: 'pro', contactEmail: 'hello@example.com' },
    { upsert: true, new: true }
  );

  const sample = [
    { make: 'Toyota', model: 'Camry',    year: 2024, price: 28500, mileage: 4200,  bodyStyle: 'sedan', fuelType: 'hybrid',
transmission: 'cvt',       drivetrain: 'fwd', exteriorColor: 'Pearl White', features: ['Adaptive cruise','Heated
seats','CarPlay'] },
    { make: 'Honda',  model: 'CR-V',     year: 2023, price: 31200, mileage: 18000, bodyStyle: 'suv',   fuelType: 'gas',
transmission: 'cvt',       drivetrain: 'awd', exteriorColor: 'Modern Steel', features: ['AWD','Sunroof','Lane assist'] },
    { make: 'Ford',   model: 'F-150',    year: 2022, price: 39900, mileage: 26000, bodyStyle: 'truck', fuelType: 'gas',
```

```
transmission: 'automatic', drivetrain: '4wd', exteriorColor: 'Oxford White', features: ['Tow package','Bedliner'] },
    { make: 'Tesla',  model: 'Model 3',  year: 2024, price: 42500, mileage: 1100,  bodyStyle: 'sedan', fuelType: 'electric',
transmission: 'automatic', drivetrain: 'rwd', exteriorColor: 'Midnight Silver', features: ['Autopilot','Glass roof'] },
    { make: 'Mazda',  model: 'CX-5',     year: 2023, price: 27800, mileage: 12500, bodyStyle: 'suv',   fuelType: 'gas',
transmission: 'automatic', drivetrain: 'awd', exteriorColor: 'Soul Red',     features: ['Leather','BOSE audio'] },
    { make: 'BMW',    model: '330i',     year: 2022, price: 36900, mileage: 29000, bodyStyle: 'sedan', fuelType: 'gas',
transmission: 'automatic', drivetrain: 'rwd', exteriorColor: 'Alpine White', features: ['Premium pkg','HUD'] },
  ];
  const docs = sample.map(s => ({ ...s, showroom: showroom._id, status: 'available', description: `${s.year} ${s.make} ${s.model}
in excellent condition. One owner, full service history.` }));
  await Car.insertMany(docs);
  res.json({ ok: true, seeded: true, count: docs.length });
});

module.exports = router;
```

DEMO FILE frontend/src/pages/Admin.jsx

A single page that's the operator's cockpit: a Seed button for empty inventories, a 30-day analytics summary, and a recent-activity table. Three views, one screen, zero ceremony. The production parents (Prigmar's `SuperAdmin` page) extend this with cross-tenant companies list, package management, and impersonation.

```
import React, { useEffect, useState } from 'react';
import { Link } from 'react-router-dom';
import client, { endpoints } from '../api/client';

export default function Admin() {
  const [summary, setSummary] = useState(null);
  const [activity, setActivity] = useState([]);
  const [busy, setBusy] = useState(false);
  const [msg, setMsg] = useState('');

  const load = async () => {
    try {
      const [s, a] = await Promise.all([
        client.get(endpoints.analytics.summary),
        client.get(endpoints.analytics.activity, { params: { limit: 20 } }),
      ]);
      setSummary(s.data);
      setActivity(a.data.items);
    } catch (e) {
      setMsg(e.response?.data?.error || e.message);
    }
  };
  useEffect(() => { load(); }, []);

  const seed = async () => {
    setBusy(true); setMsg('');
    try {
      const { data } = await client.post(endpoints.admin.seed);
      setMsg(data.seeded ? `Seeded ${data.count} cars.` : `Already seeded.`);
      load();
    } catch (e) { setMsg(e.response?.data?.error || e.message); }
    setBusy(false);
  };

  return (
    <section className="section">
      <h1>Admin</h1>
```

```
      <div className="card">
        <h2>Quick navigation</h2>
        <ul className="link-list">
          <li><Link to="/admin/users">→ Users</Link></li>
          <li><Link to="/admin/invites">→ Invites</Link></li>
          <li><Link to="/admin/roles">→ Roles &amp;amp; permissions</Link></li>
          <li><Link to="/account/api-keys">→ Your API keys</Link></li>
          <li><Link to="/docs">→ Module documentation</Link></li>
        </ul>
      </div>

      {summary && (
        <div className="card">
          <h2>Last 30 days</h2>
          <div className="stat-row">
            <div className="stat"><div className="stat-num">{summary.inventory.total}</div><div className="stat-label">Total
inventory</div></div>
            <div className="stat"><div className="stat-num">{summary.inventory.available}</div><div className="stat-
label">Available</div></div>
            <div className="stat"><div className="stat-num">{summary.leads.new}</div><div className="stat-label">New leads</div>
</div>
            <div className="stat"><div className="stat-num">{summary.leads.converted}</div><div className="stat-
label">Converted</div></div>
            <div className="stat"><div className="stat-num">{summary.signups}</div><div className="stat-label">New signups</div>
</div>
          </div>
        </div>
      )}

      <div className="card">
        <h2>Demo data</h2>
        <button className="btn-primary" onClick={seed} disabled={busy}>{busy ? 'Seeding…' : 'Seed demo inventory'}</button>
        {msg && <p className="status">{msg}</p>}
      </div>

      <div className="card">
        <h2>Recent activity</h2>
        <table className="data-table">
          <thead><tr><th>When</th><th>Actor</th><th>Action</th><th>Target</th></tr></thead>
          <tbody>
            {activity.length === 0 && <tr><td colSpan="4" className="muted">No activity yet.</td></tr>}
            {activity.map(a => (
              <tr key={a._id}>
                <td>{new Date(a.createdAt).toLocaleString()}</td>
                <td>{a.actor?.email || '—'}</td>
```

```
                <td>{a.action}</td>
                <td>{a.target} {a.targetId ? `#${String(a.targetId).slice(-6)}` : ''}</td>
              </tr>
            ))}
          </tbody>
        </table>
      </div>
    </section>
  );
}
```

DEMO FILE frontend/src/pages/Inventory.jsx

The customer-facing CRUD list. Search, filter, paginate, click into detail. ~120 lines. This is what every product list looks like in every SaaS — the only thing that changes is what the rows represent. The graduation move is saving filter combos as named `FilterView`s — see chapter narrative.

```
import React, { useEffect, useState } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { fetchCars } from '../actions/carActions';
import CarCard from '../components/CarCard';

export default function Inventory() {
  const dispatch = useDispatch();
  const { items, total, loading } = useSelector(s => s.cars);
  const [filters, setFilters] = useState({ q: '', bodyStyle: '', fuelType: '', maxPrice: '' });

  useEffect(() => { dispatch(fetchCars({ ...filters, limit: 24 })); }, [dispatch, filters]);

  const set = (k) => (e) => setFilters(f => ({ ...f, [k]: e.target.value }));

  return (
    <section className="section">
      <h1>Inventory</h1>
      <div className="filters">
        <input placeholder="Search make, model, keywords" value={filters.q} onChange={set('q')} />
        <select value={filters.bodyStyle} onChange={set('bodyStyle')}>
          <option value="">Any body style</option>
          {['sedan','suv','truck','coupe','convertible','hatchback','wagon','van'].map(b => <option key={b} value={b}>{b}
</option>)}
        </select>
        <select value={filters.fuelType} onChange={set('fuelType')}>
          <option value="">Any fuel</option>
          {['gas','diesel','hybrid','electric'].map(b => <option key={b} value={b}>{b}</option>)}
        </select>
        <input type="number" placeholder="Max price" value={filters.maxPrice} onChange={set('maxPrice')} />
      </div>
      <p className="muted">{loading ? 'Loading…' : `${total} car${total === 1 ? '' : 's'} match.`}</p>
      <div className="grid grid-3">
        {items.map(c => <CarCard key={c._id} car={c} />)}
      </div>
    </section>
```

```
  );
}
```

CHAPTER 09 · THE FREE BUILDER COURSE

Analytics & Measurement — What You Measure Is What You Improve

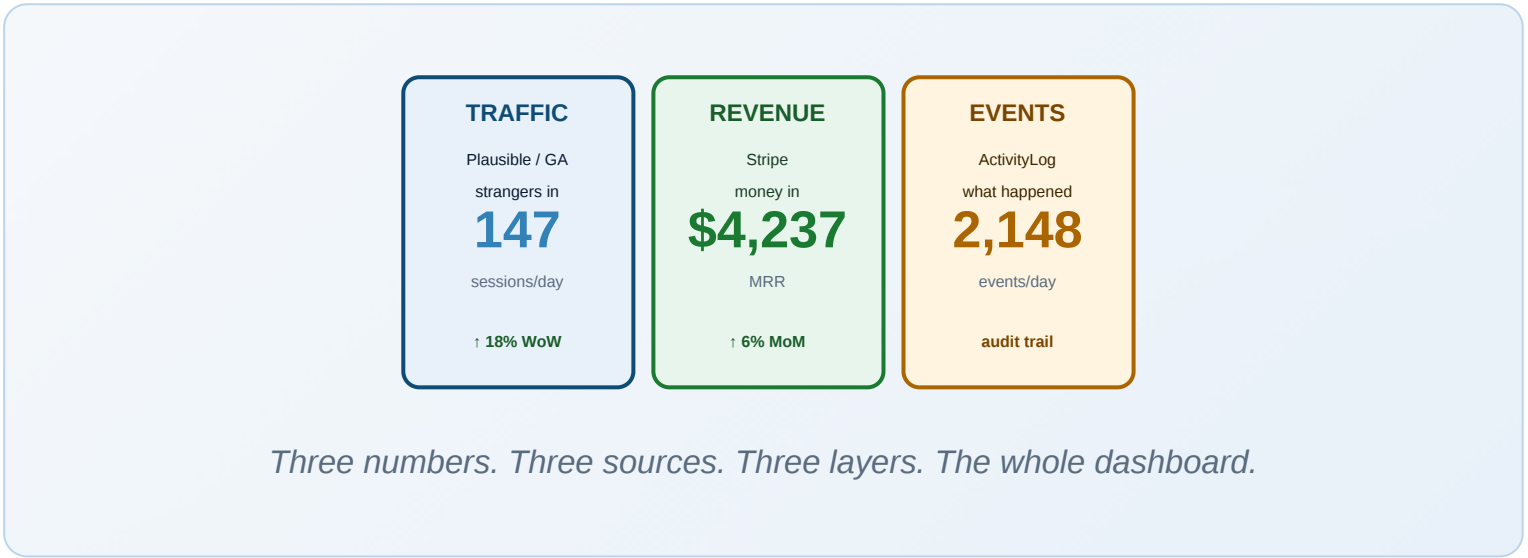
If you can't answer how many signups this week in under 30 seconds, you don't have analytics — you have vibes.

> *There are two ways to run a SaaS. The first is on a hunch. The second is with three numbers on a screen that update every time you refresh. Only one of these scales past Tuesday.*

The three layers (this is the whole framework)

- **Traffic** — sessions, sources, top landing pages. Where strangers come from.
- **Revenue** — MRR, ARR, new vs expansion vs churn. The number on the wall.
- **Events** — the audit trail. Every meaningful action: signup, paywall hit, button click. Your time machine when something breaks.

Three layers. Three sources. Most operators tangle them or skip two of them. Don't.



Three numbers. Three sources. Three layers. The whole dashboard.

The activity log is your time machine

You will have customers who say "it broke when I clicked X." You will not believe them. You will not have screen recordings. What you will have is the activity log: every meaningful action with `actor`, `action`, `target`, `diff`, and timestamp. Open the log, find their last 10 events, see exactly what happened, fix the bug, sleep.

> ⏰ **THE 2AM RULE** If your activity log is missing a key event, you will discover this fact at 2 AM, when a paying customer's data is in an unknown state and you have nothing to roll back to. Log every action you can imagine ever needing. Storage is cheap; sleep is not.

The one-page dashboard

The admin Dashboard in the demo is one page. Eight numbers, one chart, one recent-activity table. That's it. Most operators build a 40-page analytics tower and then never look at any page beyond the first. Build the first page well, then stop.

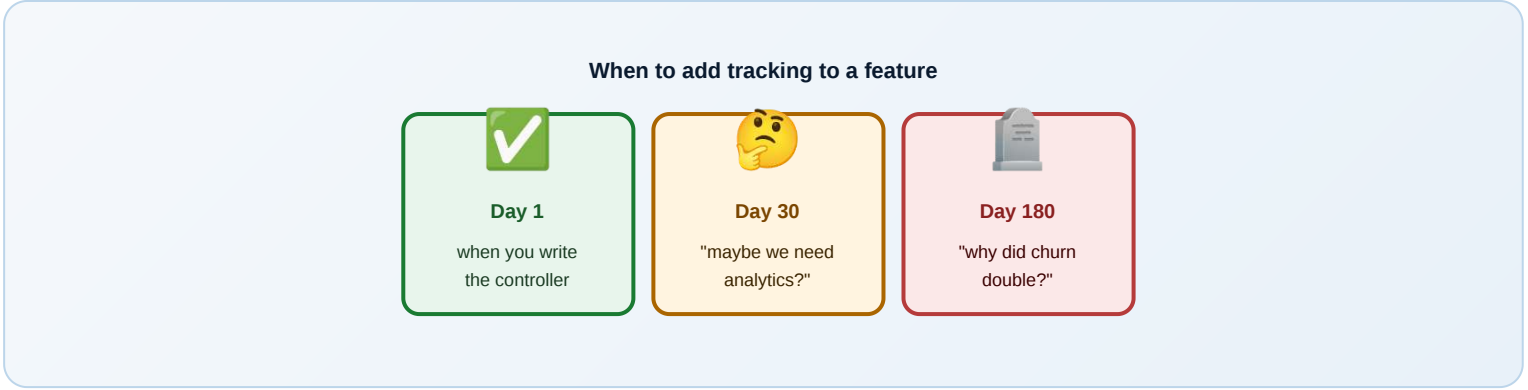
```
// Run all five queries in parallel — concurrency is free.
const [users, leads, cars, revenue, recent] = await Promise.all([
  User.countDocuments(),
  Lead.countDocuments({ status: 'new' }),
  Car.countDocuments({ status: 'available' }),
  computeMRR(),
  ActivityLog.find().sort({ createdAt: -1 }).limit(20),
]);
```

Sequential, this is 250ms. `Promise.all`, it's 50ms. Concurrency is free; use it. Your dashboard should be instant.

Tracking on Day One (backfilling is impossible)

Add the `activityLogs.record(...)` call to every controller from the day you write it. Login, signup, paywall hit, plan upgrade, plan downgrade, refund, password change. You do not need to think about which events matter yet. You need *volume*. In year two, when you finally know what to look for, you'll find it in the haystack you've been quietly building.

Trying to add tracking *after* a feature is in production is approximately impossible. The team that didn't track signups in month 1 will not know which of their first 200 users came from Reddit in month 4. They will guess. Their guesses will be wrong.



▸ KEY TAKEAWAY

Three numbers on a screen — traffic, revenue, events — refreshable in 30 seconds. Backfilling analytics is impossible. Track everything from Day 1. Run the dashboard queries in parallel. Stop building the second page.

The deep analytics layer (when "three numbers" stops being enough)

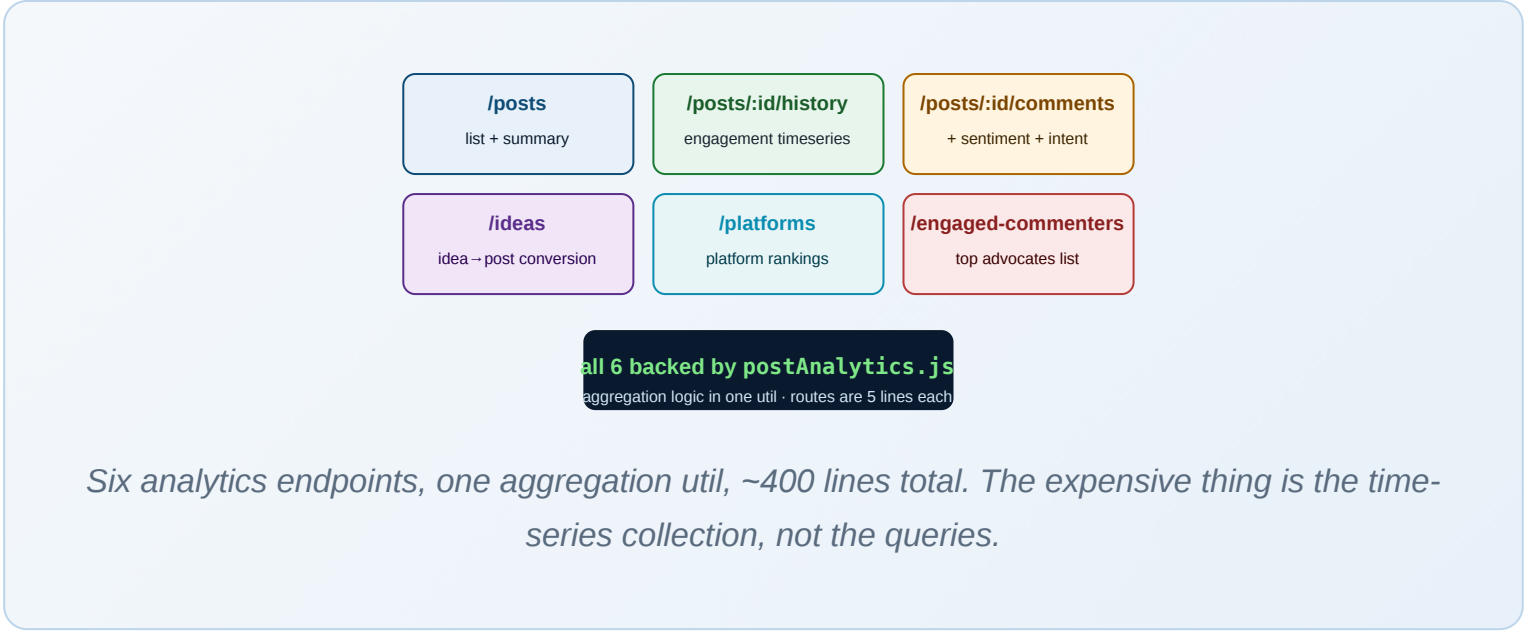
The one-page dashboard is right for the first six months. Once you have customers using the product daily, they want their *own* analytics — and your customer-success team wants to see what's working and what isn't. This is where the framework grows a real measurement layer.

Engineering-style work tracking (BidLight pattern)

If your core service involves billable work — engineering hours, design hours, agency retainers — you need a per-project per-revision tracker. BidLight ships `EngineeringTracking`: rows that capture `{ user, project, revision, hours, billable, createdAt }` and four endpoints to read them by company, project, or revision. The aggregation page is a simple grouped sum. You wouldn't think it's analytics. It is, because it's the thing the customer pays for.

Content-engagement analytics (Prigmar's PostAnalytics)

If your product produces content — blog posts, social media posts, ads — you'll end up building Prigmar-style engagement analytics. Six endpoints, each backed by a util in `postAnalytics.js`:



Six analytics endpoints, one aggregation util, ~400 lines total. The expensive thing is the time-series collection, not the queries.

The trick is the `PostAnalyticsSnapshot` model: at fixed intervals (every hour for new posts, daily for old), a cron job fetches engagement metrics from each platform (likes, comments, shares, score) and writes one row per post per timestamp. Querying becomes a simple `find().sort({ts: 1})`. The engagement-over-time chart is one query.

Sentiment + intent classification (the AI layer)

When you have comments coming in from social platforms, you want to know: is this praise, complaint, question, spam? Prigmar adds two columns to `SocialComment`: `sentiment` (positive/neutral/negative) and `intent` (praise/complaint/question/spam/other), both populated by an LLM call when the comment arrives. The PostAnalytics dashboard renders them as colored chips.

Cost: ~$0.0002 per comment with Haiku. Value: customer support filters to "complaints" before the angry tweet goes viral.

The LLM cost analytics surface (when you have LLM calls in product)

If you use LLMs inside the product (sentiment, classification, generation), you'll want visibility into spend before your provider sends a "you exceeded $5K this month" email. Prigmar's `llmBudget` module ships:

- `llmCostSnapshot` — daily aggregated cost (one row per company per day per provider)
- `llmProviderUsage` — per-request usage rows (model, tokens-in/out, cost-cents) for the granular view
- Endpoints: `/llmBudget`, `/llmBudget/series`, `/llmBudget/projection`, `/llmBudget/companies`, `/llmBudget/provider-truth`, `/llmBudget/markup`, `/llmBudget/panic` (the emergency-stop button)
- Frontend: `AdminCostControl` — line charts of daily cost, projection-to-month-end, per-company breakdown

The "panic" endpoint is the kind of feature you don't appreciate until you appreciate it: it sets a hard cap, and any LLM-touching middleware reads it and short-circuits. **Build this before your first viral moment, not after.**

▸ KEY TAKEAWAY, EXPANDED

One-page dashboard for v1. Customer-facing analytics (work tracking, content engagement) when your customers ask for them. AI sentiment + intent when you have comments to triage. LLM cost analytics + a panic button when you have LLM calls. Each layer earns its keep after the previous one has been used in anger.

ANALYTICS SETUP

Measure What Matters

We'll set up

- event tracking
- analytics dashboards
- user behavior logs

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE back/routes/analytics.js

Two endpoints: `/summary` (numbers your manager would ask for) and `/activity` (audit trail). The `Promise.all` in `/summary` is important — running the five count queries sequentially turns a 50ms dashboard into a 250ms dashboard. Concurrency is free. Production parents extend this with PostAnalytics (6 endpoints) and LLM budget (7 endpoints) — see chapter narrative.

```
// Three views: traffic-style counts, revenue summary, recent activity.
// Staff-only — anyone with billing data shouldn't be on the public site.
const router = require('express').Router();
const requireAuth = require('../middlewares/requireAuth');
const requirePermission = require('../middlewares/requirePermission');
const Car = require('../models/Car');
const Lead = require('../models/Lead');
const User = require('../models/User');
const ActivityLog = require('../models/ActivityLog');

router.use(requireAuth, requirePermission(['manager','admin']));

router.get('/summary', async (req, res) => {
  const since = new Date(Date.now() - 30 * 24 * 60 * 60 * 1000);
  const [carsTotal, carsAvailable, leadsNew, leadsConverted, signups] = await Promise.all([
    Car.countDocuments({}),
    Car.countDocuments({ status: 'available' }),
    Lead.countDocuments({ status: 'new', createdAt: { $gte: since } }),
    Lead.countDocuments({ status: 'converted', createdAt: { $gte: since } }),
    User.countDocuments({ createdAt: { $gte: since } }),
  ]);
  res.json({
    ok: true,
    range: 'last_30_days',
    inventory: { total: carsTotal, available: carsAvailable },
    leads: { new: leadsNew, converted: leadsConverted },
    signups,
  });
});

router.get('/activity', async (req, res) => {
  const limit = Math.min(Number(req.query.limit || 50), 200);
  const items = await ActivityLog.find().sort({ createdAt: -1 }).limit(limit).populate('actor', 'email name');
  res.json({ ok: true, items });
});
```

```
module.exports = router;
```

DEMO FILE back/utils/activityLogs.js

Two helpers, ~30 lines. The key idea: **logging must never break the request**. The try/catch around `ActivityLog.create` swallows failures intentionally — a logging bug should never take down a customer's request.

```
// Two helpers:
//   record({ actor, action, target, ... }) — fire-and-forget log a domain event.
//   requestRecorder() — middleware that attaches req.startedAt and logs slow/important requests.
const ActivityLog = require('../models/ActivityLog');

async function record({ actor, showroom, action, target, targetId, diff, req }) {
  try {
    await ActivityLog.create({
      actor,
      showroom,
      action,
      target,
      targetId,
      diff,
      ip: req ? (req.ip || req.connection?.remoteAddress) : null,
      ua: req ? req.get('user-agent') : null,
    });
  } catch (e) {
    // Logging must never break the request — swallow.
    console.warn('[activityLogs.record] failed:', e.message);
  }
}

function requestRecorder() {
  return (req, res, next) => {
    req.startedAt = Date.now();
    next();
  };
}

module.exports = { record, requestRecorder };
```

DEMO FILE back/models/ActivityLog.js

Five indexed fields: `actor`, `showroom`, `action`, `createdAt`, plus `diff` as Mixed for ad-hoc payloads. Indexes are the difference between a dashboard that loads in 50ms and one that loads in 5 seconds when your log has 10M rows. Add them on Day 1.

```
// Every meaningful action — login, listing edit, status change — lands here.
// This is the audit log; it's also analytics fuel.
const mongoose = require('mongoose');

const ActivityLogSchema = new mongoose.Schema({
  actor:    { type: mongoose.Schema.Types.ObjectId, ref: 'User', index: true },
  showroom: { type: mongoose.Schema.Types.ObjectId, ref: 'Showroom', index: true },
  action:   { type: String, required: true, index: true },   // e.g. 'car.create', 'lead.status_change'
  target:   { type: String },                                 // collection name or URL
  targetId: { type: mongoose.Schema.Types.ObjectId },
  diff:     { type: mongoose.Schema.Types.Mixed },
  ip:       String,
  ua:       String,
  createdAt:{ type: Date, default: Date.now, index: true },
});

module.exports = mongoose.model('ActivityLog', ActivityLogSchema);
```

The Core Service — One Object, A Hundred Variations

Define your core object cleanly and the rest of the product writes itself. Get it wrong and you'll rewrite endpoints for three years.

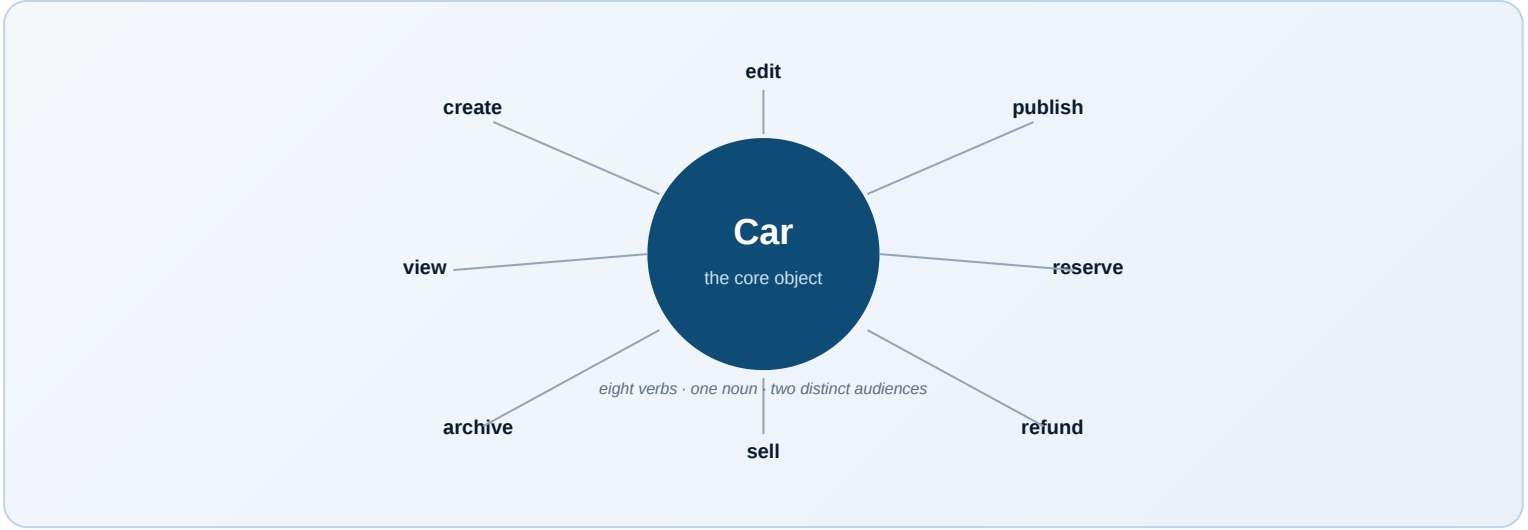
> *Every successful SaaS resolves into a single core object that the customer creates and manipulates. BidLight has Project. Prigmar has Brand. Tutoplus has Course. Tesla DGF has Document. The auto showroom has Car. Yours has a thing. Name it on a whiteboard before you write a line.*

The whiteboard exercise (do this first)

Before you touch a keyboard, write down:

- **What is the noun?** Singular. Specific. "Car listing", not "automotive entity".
- **What does a customer do to it?** Verbs. Create, edit, publish, share, sell, archive.
- **What does the operator do to it?** Different verbs. Review, approve, audit, refund, undelete.
- **What are its life-cycle states?** draft → available → reserved → sold → archived. Or whatever yours is.
- **What hangs off it?** Leads, comments, photos, prices, history.

That whiteboard is your data model. Don't draw entity-relationship diagrams. Draw a noun in a circle and verbs around it. The schema falls out.



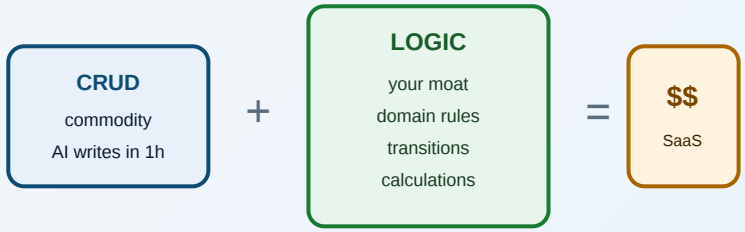
📚 **FREE ANTI-PATTERN** Don't model six "kinds of things" with sibling tables. *Cars*, *Vans*, *Bikes*, *Boats*, *Aircraft*, *Hovercraft* — six tables, five duplicate controllers, six different filter pages. Or: *Listings*, with a `type` field. One table, one controller, one page. Choose the second one. (Yes I am pretending no one ever sells a hovercraft on Auto Showroom.)

CRUD plus logic — the differentiator

CRUD is the floor. Anyone can build CRUD; your AI worker can write it before lunch. The thing that makes *your* product is the **logic**: the rules, transitions, automations, calculations that wrap the CRUD. Two examples:

- BidLight's "construction-project" CRUD is trivial. Its logic — auto-computing material costs from a 2D takeoff against a regional price book — is the moat.
- Prigmar's "brand" CRUD is trivial. Its logic — generating SEO-optimized articles from a brand profile, scheduling them, measuring traffic — is what you pay for.

The car-sales demo deliberately has *thin* logic — view-count increment, status transitions, search. Just enough to show the pattern. Your real product will have ten times more logic; the CRUD will look identical.



CRUD is the floor. Logic is the elevator. Build both; charge for the elevator.

Status fields are the secret weapon

Notice the `status` enum on the Car schema: `draft`, `available`, `reserved`, `sold`, `archived`. Five values. Each one gates different actions: a `sold` car can't be reserved; a `draft` car doesn't show in public listings. The transitions are the rules; the rules are the product.

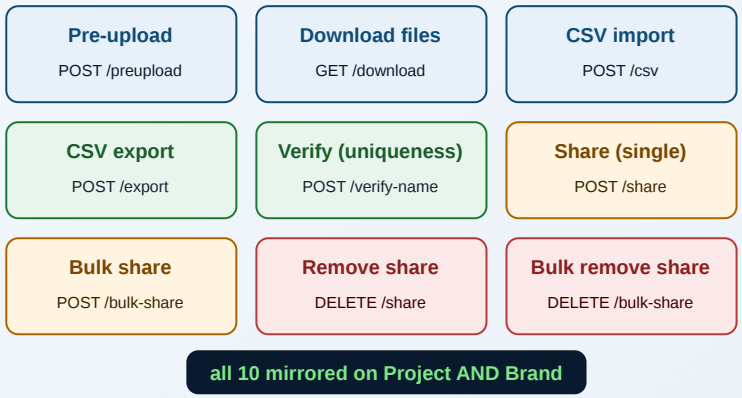
Most beginners model state with three booleans (`isPublished`, `isSold`, `isArchived`). Don't. You end up in invalid combinations (`isSold && isArchived && !isPublished` = ???). One enum, transition rules, peace.

▸ KEY TAKEAWAY

Whiteboard the noun before you touch a keyboard. CRUD is the floor; logic is the moat. Use status enums, not boolean soup. Get this layer right and the product writes itself.

The file/share/import/export surface (the 10 endpoints beyond basic CRUD)

Five CRUD endpoints get you to "the customer can manage their data." Ten more endpoints get you to "the customer can actually *use* the data with their team." Both BidLight (Projects) and Prigmar (Brands) ship the same surface, only the entity name differs:



Ten endpoints. Same pattern in both production codebases. Different noun, identical surface.

Why pre-upload + download are separate routes

Naive design: `POST /api/cars/:id/files` with the file in the body. The customer's browser uploads to your API server, which writes to disk, which proxies to S3. Three hops, your server holds the file twice in memory, large files time out.

Production design (BidLight, Prigmar pattern):

1. `POST /preupload` — server generates a signed S3 PUT URL, returns it.
2. Browser uploads directly to S3 with the signed URL (no server proxy).
3. Browser POSTs the resulting S3 key back to the regular update endpoint.
4. `GET /download` — server generates a signed GET URL (5-minute TTL) and redirects to it.

The server never holds the file. Bandwidth cost on your VPS drops to near zero. The customer can upload 500MB without timing out. **Adopt this pattern from Day 1.** Adding it later means rewriting every upload form in the app.

Bulk share = the team-feature that closes deals

Single-share is easy: "Joe, here's the link to this project." Bulk share is the killer: "Select these 12 projects, add Acme Co. as a viewer with read-only access until Friday, send the invite." Without bulk share, your sales rep does this twelve times by hand and quietly stops selling to teams.

The endpoint takes an array of resource IDs plus a share payload. The controller iterates, calls the single-share logic for each, returns a summary `{ successCount, failed: [...] }`. Twelve lines on top of the existing single-share. Massive UX improvement.

Brand settings (the kitchen-sink config per core object)

Each core object eventually grows its own settings drawer. Prigmar's `BrandSettings` holds: voice/tone preferences, target audience descriptors, default hashtags, posting schedule, approval workflow flags, integration credentials per platform. CRUD in three endpoints. The settings table is the place to put per-core-object config that doesn't deserve its own first-class model.

The 'satellites' that ride the core object

The core object always grows satellites — child entities that only make sense in its context. In BidLight: `Milestone`, `Revision`, `DocControl` (sheet drawings), `Discipline`, `Room` (collaboration thread). Each satellite gets its own CRUD-plus-bulk-plus-CSV (7 endpoints each: create, list, get, update, delete, bulk-delete, csv-import).

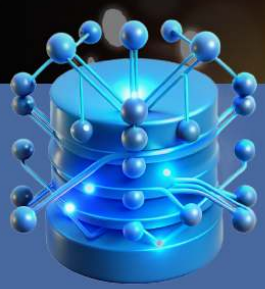
Same pattern repeats forever: noun, status enum, six CRUD endpoints, bulk + CSV. Once you've built this skeleton for one satellite, you've built it for all of them. Your AI worker can scaffold a new one in 20 minutes.

▸ KEY TAKEAWAY, EXPANDED

5 CRUD endpoints + 10 file/share/import/export endpoints + a settings drawer + 1-N satellite models, each with the same 7-endpoint shape. That's the full core-service surface. Memorize the shape; your AI worker can spin up a new entity at this template in under an hour.

THE CORE SERVICE

What You Actually Sell



Definition of the Core Service

- The central, functional part of your SaaS product.
- It's what your customers are actually paying for.

Value Delivery

- The mechanism through which your SaaS converts a user's problem into a tangible solution.
- It should clearly demonstrate the transformation from pain point to outcome.

Diagram

- Input: User's problem or need.
- Your SaaS: The system/process/feature set that performs the work.
- Output: The value-driven result or desired outcome.

Purpose of this Module

- To help users identify and articulate the true function and benefit of their product.
- To ensure the core service aligns with customer expectations and needs.

DEFINE YOUR CORE OBJECT

What Is Your SaaS Actually Managing?

01

Identify the Core Object

- Every SaaS product revolves around a primary object or data entity.
- Examples include: Projects, Documents, Videos, Analytics, Bookings.

02

The Core Object Shapes Your Platform

- Your core object dictates the structure of your features and workflows.
- It's the "thing" your users come to manage, improve, or engage with.

03

Your Unique Value Stems From This Object

- The way your SaaS handles, enhances, or interacts with this object is what sets you apart from competitors.
- Innovation often comes from rethinking how the object is used or presented.

04

Everything Ties Back to the Core Object

- User roles, permissions, dashboards, and integrations all center on this object.
- Understanding it deeply is essential for UX design, development, and marketing.

BUILD THE CORE FLOW

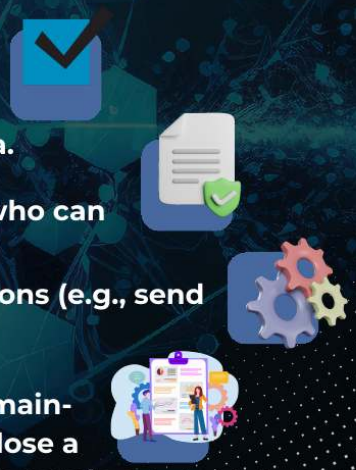
CRUD + Logic = Service

Why CRUD Matters?

- Create: Add a new record, like a task, post, or project.
- Read: View data in dashboards, lists, or detail pages.
- Update: Modify existing records, such as editing a task's due date.
- Delete: Remove data when it's no longer needed.

Where Logic Comes In

- Validation: Prevent users from entering invalid data.
- Permissions: Determine who can do what.
- Automations: Trigger actions (e.g., send emails, update statuses).
- Business Rules: Apply domain-specific logic (e.g., can't close a ticket without a resolution note).



THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE back/models/Car.js

The core object for this demo. Read the schema as a list of the questions a customer can ask about a car. Every field is either a customer-facing fact (price, mileage, features) or operator-facing metadata (status, views, createdBy). When you can't decide whether a field belongs on the core object, ask: would a customer ever ask about it? Production parents (BidLight Project, Prigmar Brand) follow the same shape with more satellites — see chapter narrative.

```
// The core domain object. Every other resource (leads, photos, comments) hangs off this.
const mongoose = require('mongoose');

const CarSchema = new mongoose.Schema({
  showroom:  { type: mongoose.Schema.Types.ObjectId, ref: 'Showroom', index: true },
  make:      { type: String, required: true, index: true },
  model:     { type: String, required: true, index: true },
  year:      { type: Number, required: true, min: 1900 },
  trim:      String,
  price:     { type: Number, required: true, min: 0 },
  mileage:   { type: Number, default: 0 },
  vin:       { type: String, index: { unique: true, sparse: true } },
  bodyStyle: { type: String, enum: ['sedan','suv','truck','coupe','convertible','hatchback','wagon','van'] },
  fuelType:  { type: String, enum: ['gas','diesel','hybrid','electric'] },
  transmission: { type: String, enum: ['automatic','manual','cvt'] },
  drivetrain:{ type: String, enum: ['fwd','rwd','awd','4wd'] },
  exteriorColor: String,
  interiorColor: String,
  features:  [String],
  photos:    [String],   // URLs
  description: String,
  status:    { type: String, enum: ['draft','available','reserved','sold','archived'], default: 'available', index: true },
  views:     { type: Number, default: 0 },
  createdBy: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now },
});

CarSchema.pre('save', function (next) { this.updatedAt = new Date(); next(); });
CarSchema.index({ make: 1, model: 1, year: -1 });

module.exports = mongoose.model('Car', CarSchema);
```

DEMO FILE back/controllers/car.ctrl.js

The full CRUD + a sprinkle of logic. `list` handles filter shape and `$or` regex for search. `get` increments view count fire-and-forget. `create` and `update` validate, then log to the activity feed. Read this end-to-end — it's the canonical shape every other resource in the app follows.

```
// CRUD for the core object. Public list/get; create/update/delete are staff-only.
const Car = require('../models/Car');
const activityLogs = require('../utils/activityLogs');

exports.list = async (req, res) => {
  const { status, make, bodyStyle, minPrice, maxPrice, q, limit = 24, skip = 0 } = req.query;
  const filter = {};
  if (status) filter.status = status; else filter.status = 'available';
  if (make) filter.make = make;
  if (bodyStyle) filter.bodyStyle = bodyStyle;
  if (minPrice || maxPrice) {
    filter.price = {};
    if (minPrice) filter.price.$gte = Number(minPrice);
    if (maxPrice) filter.price.$lte = Number(maxPrice);
  }
  if (q) filter.$or = [
    { make: new RegExp(q, 'i') }, { model: new RegExp(q, 'i') }, { description: new RegExp(q, 'i') }
  ];
  const items = await Car.find(filter).sort({ createdAt: -1 }).skip(Number(skip)).limit(Math.min(Number(limit), 100));
  const total = await Car.countDocuments(filter);
  res.json({ ok: true, items, total });
};

exports.get = async (req, res) => {
  const car = await Car.findById(req.params.id);
  if (!car) return res.status(404).json({ ok: false, error: 'Not found.' });
  Car.updateOne({ _id: car._id }, { $inc: { views: 1 } }).catch(() => {});
  res.json({ ok: true, car });
};

exports.create = async (req, res) => {
  const car = await Car.create({ ...req.body, createdBy: req.user._id });
  activityLogs.record({ actor: req.user._id, action: 'car.create', target: 'Car', targetId: car._id, req });
  res.status(201).json({ ok: true, car });
};

exports.update = async (req, res) => {
```

```
  const car = await Car.findById(req.params.id);
  if (!car) return res.status(404).json({ ok: false, error: 'Not found.' });
  Object.assign(car, req.body);
  await car.save();
  activityLogs.record({ actor: req.user._id, action: 'car.update', target: 'Car', targetId: car._id, diff: req.body, req });
  res.json({ ok: true, car });
};

exports.remove = async (req, res) => {
  await Car.deleteOne({ _id: req.params.id });
  activityLogs.record({ actor: req.user._id, action: 'car.delete', target: 'Car', targetId: req.params.id, req });
  res.json({ ok: true });
};
```

CHAPTER 11 · THE FREE BUILDER COURSE

Billing — Stripe Will Run Your Business If You Let It

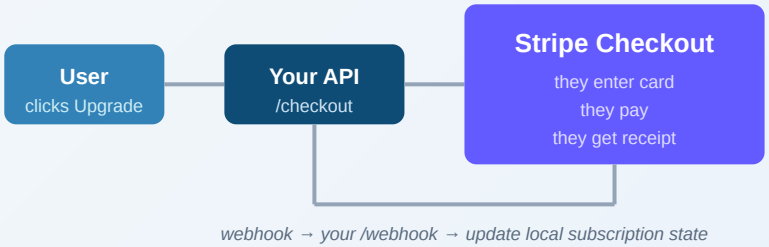
Webhooks are the source of truth, not your database. Verify, then mirror. And do not, do not, build your own Customer Portal.

> *Subscription billing is what makes SaaS compound. It is also the place where solo operators most often shoot themselves in the foot, both feet, and somehow a third foot that should not exist. Stripe is the gun. Use it on the targets, not the feet.*

Three endpoints. That's the integration.

- `GET /api/billing/plans` — return the plan catalog. Frontend renders.
- `POST /api/billing/checkout` — create a Stripe Checkout session, return the URL, frontend redirects.
- `POST /api/billing/webhook` — verify signature, switch on event type, update your local subscription state.

That's it. Three endpoints. Stripe runs the entire checkout UI, the entire upgrade/downgrade UI, the entire failed-payment retry logic, the entire VAT/tax compliance. You run the part where you decide who sees what.



Three endpoints. Stripe does the heavy lifting. You do the gate-keeping.

The webhook is the source of truth

This is the rule that separates working billing integrations from haunted ones: **your database is a mirror of Stripe, not the other way around.** When the checkout succeeds, do NOT update your database from the frontend "success" callback. Wait for the webhook. The webhook is the truth. The frontend is a hint.

> 🪤 **TRAP** Update on frontend success: works perfectly in testing. In production, the user closes the tab before redirect, never hits your success page, pays Stripe successfully — and your DB still thinks they're on Starter. Refunds. Bug reports. Lost weekend. Do not update on success. Update on webhook.

Plan gating uses your existing capability system

Plan tier is just another input to the capability check. `analytics.advanced` is a capability. The `Pro` plan bundle includes it. The `Starter` plan doesn't. The middleware is the same middleware you use for roles. Don't build a second permission system for billing; use the one you already have.

Stub mode — the secret to fast iteration

The demo's `billing.js` opens with this:

```
const stripe = config.stripeSecret
  ? require('stripe')(config.stripeSecret)
  : null;

router.post('/checkout', requireAuth, async (req, res) => {
  if (!stripe) {
    // Stub mode — fake URL for demos & reviews
    return res.json({ ok: true, url: `/billing/demo-checkout?plan=${plan}` });
  }
  /* real Stripe call */
});
```

This pattern is invaluable. When `STRIPE_SECRET_KEY` is empty (every reviewer, every screenshot session, every demo call), the API returns a plausible-looking fake URL. The frontend flow is fully demoable. You don't have to wire test keys into your demo environment, and you can't accidentally charge yourself.

One env-var flag. Two complete realities. This kind of toggle is what makes the difference between a codebase you can demo and one you can't.

The eight rules of billing

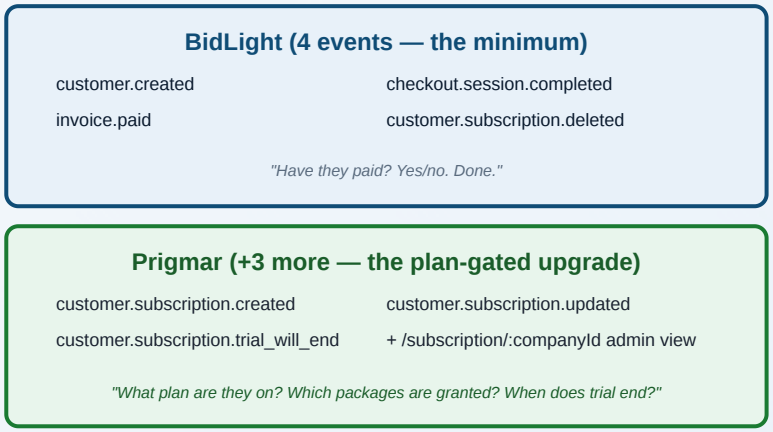
- Stripe Customer Portal handles 80% of billing UI. Do not build your own.
- Webhooks are source of truth. Always verify the signature.
- Mirror subscription state on your User (or Workspace) doc.
- Plan tier is a capability bundle. Use the existing middleware.
- Log every Stripe event to your activity log.
- Idempotency keys on customer-facing actions.
- Test failed payments — the retry/dunning flow is where customers churn silently.
- VAT/tax — turn on Stripe Tax. Pay the 0.5%. Don't think about it again.

▸ KEY TAKEAWAY

Three endpoints, one webhook, one capability bundle per plan. Stripe runs the checkout; you run the gating. Stub mode for demos. The webhook is the source of truth. Anything else is yak-shaving you can do at $50K MRR.

The webhook events that actually matter (and which ones earn their keep)

The Stripe webhook docs list about 200 events. You will never handle 200. BidLight's billing controller handles four. Prigmar's handles seven. The difference is everything you need to know about graduating from "I take subscriptions" to "I run a plan-gated SaaS."



Four events get you paid; seven events let you run a plan-tier business. The delta is where plan-gating, package sync, and trial reminders live.

What each extra event unlocks

- `customer.subscription.created` — fired when a subscription begins. This is your hook for **granting packages**: look at the Stripe price ID, find the matching plan metadata, copy the capability bundle to `company.permissions`. Without this, customers complete checkout but their plan-gated features stay locked until the next `invoice.paid` 5 minutes later. That 5-minute gap is one of the most-emailed customer-support tickets of all time.
- `customer.subscription.updated` — fired on plan change (upgrade, downgrade, addon). Your hook for **recomputing the package set**. Customer was Pro yesterday, downgraded to Starter today; `analytics.advanced` needs to drop off. This event is the one that keeps your plan-gating honest.
- `customer.subscription.trial_will_end` — fired 3 days before a trial ends. Your hook for **trial-reminder email + in-app banner**. Don't email manually; emit a `TRIAL_REMINDER_REQUESTED` event to your outbox worker, which handles retries, rate limiting, and delivery.

Package-sync (the function the chapter wouldn't be complete without)

The Prigmar trick: **capability bundles live on the Stripe price metadata**, not in your application code. Each Stripe price has a `metadata.packages` field like `"analytics.basic,car.delete,user.invite"`. When a subscription event fires, you call `resolvePackagesFromSubscription(sub)`: it reads the items, expands each price's metadata, dedupes, and returns the full capability set.

Why on Stripe metadata, not in your code? **Because the source of truth for "what does this plan include" is the place customers see the price**. Storing it elsewhere creates a sync problem. Two helper functions — `grantPackages(company, set)` and `revokePackages(company, set)` — handle the diff. Total: ~80 lines.

The `requireQuota` middleware closes the loop with Ch 7

Now that packages live on `company.permissions`, your routes can check them. `requireQuota(capability)` (covered in Ch 7) reads `company.permissions`, checks for the capability key, returns 403 + an "upgrade required" envelope if missing. **Permissions and billing become one system**: the role layer answers "can your user do this within your plan?", the quota layer answers "does your plan include this?".

Super-admin subscription tooling (when customers email "I need a one-month extension")

Production reality: customers will ask for plan overrides outside of Stripe's UI. "Can you give me Pro features until Tuesday?" "We're switching from monthly to annual and need the credit applied manually." Build the super-admin tooling for this:

- `GET /stripe/subscription/:companyId` — read-only view of Stripe state (status, current_period_end, items, packages currently granted)
- `PUT /superAdminPanel/companies/:id/packages` — manually grant or revoke packages, bypassing Stripe. Logs to activity log. Time-bounded if you want.

Ten lines apiece. Saves you from logging into Stripe to fish for the right customer ID at 4 PM on a Friday.

The trial-reminder outbox pattern (graduation move)

When `trial_will_end` fires, the naive code sends the email synchronously inside the webhook handler. Don't. Emit an outbox event:

```
// webhook handler — keep it fast
case 'customer.subscription.trial_will_end':
  await Outbox.create({
    type: 'TRIAL_REMINDER_REQUESTED',
```

```
    payload: { companyId, trialEndDate: sub.trial_end },
    runAfter: new Date(), // now
    attempts: 0,
  });
  break;
```

A background worker picks up outbox rows, sends the email via your email provider, marks the row done. Failures retry with backoff. The webhook handler stays fast (Stripe times out at 10 seconds). Same pattern works for any "side-effect we want to retry on failure".

▸ KEY TAKEAWAY, EXPANDED

4 webhook events get you paid. Add 3 more (subscription created/updated/trial_will_end) to run plan-gated SaaS properly. Store capability bundles on Stripe price metadata. Use `requireQuota` middleware to enforce. Add super-admin override endpoints for support escalations. Use the outbox pattern for any email a webhook needs to send. Now billing is a system you can leave alone for months.

STRIPE PAYMENTS INTEGRATION

01

Multiple Payment Options

- Subscriptions: Set up recurring billing plans for monthly or annual payments. Ideal for SaaS products with tiered pricing.
- One-time Payments: Great for purchases like digital downloads, add-ons, or extra services.
- Lifetime Deals: Offer a single upfront payment for long-term access — popular for early adopters or limited-time promotions.

02

Frictionless Checkout Experience

- Stripe's hosted checkout pages are secure, mobile-friendly, and PCI-compliant out of the box.
- With a few lines of code, you can launch a polished payment experience without reinventing the wheel.

03

Integration Options

- Use Stripe Elements or Stripe Checkout to embed payment forms directly into your app.
- Stripe's APIs also let you fully customize the payment flow if you need a more branded experience.

Make It Easy to Get Paid

04

User-Centric Billing Management

- Provide users with a portal to update their payment methods, manage subscriptions, or cancel plans — all securely hosted by Stripe .
- Stripe grows with your business — from MVP to global scale with features like multi-currency support and fraud protection.

CONTROL FEATURES BY PLAN

License + Access Management

License Logic Implementation

- Core idea: Lock or unlock features based on user plan.
- Add logic to your backend (or middleware) to determine feature access.

Locking and Unlocking UI Elements

- Locked icons over inaccessible features
- Tooltips explaining required plan level
- Call-to-action buttons for upgrading

Managing License Data

- Database (e.g., PostgreSQL, Firebase)
- JWT Tokens (short-term license validation)
- API layer (checks license before executing requests)

Plan Upgrades and Downgrades

- Sync license data immediately after checkout
- Trigger feature availability refresh
- Handle edge cases (e.g., expired licenses, failed payments)

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE back/routes/billing.js

Three endpoints: list plans, start a checkout session, receive webhooks. Note the **stub mode** at the top — when STRIPE_SECRET_KEY is empty, /checkout returns a fake URL so the frontend can be demoed without real keys. This pattern is invaluable during development and code review. Production parents add the package-sync logic on subscription.created/updated and the outbox emit on trial_will_end — see chapter narrative.

```
// Stripe billing — checkout sessions + webhook verification.
// Wire your real keys in .env. Stub mode returns predictable fake URLs when STRIPE_SECRET_KEY is empty.
const router = require('express').Router();
const config = require('../bin/config');
const requireAuth = require('../middlewares/requireAuth');

const stripe = config.stripeSecret ? require('stripe')(config.stripeSecret) : null;

const PLANS = {
  starter:   { name: 'Starter',   priceId: 'price_starter_demo',    price: 29 },
  pro:       { name: 'Pro',       priceId: 'price_pro_demo',        price: 99 },
  enterprise:{ name: 'Enterprise',priceId: 'price_enterprise_demo', price: 299 },
};

router.get('/plans', (req, res) => res.json({ ok: true, plans: PLANS }));

router.post('/checkout', requireAuth, async (req, res) => {
  const { plan } = req.body || {};
  if (!PLANS[plan]) return res.status(400).json({ ok: false, error: 'Unknown plan.' });
  if (!stripe) {
    // Stub mode — return a placeholder URL so the frontend can demo the flow without keys.
    return res.json({ ok: true, url: `/billing/demo-checkout?plan=${plan}` });
  }
  try {
    const session = await stripe.checkout.sessions.create({
      mode: 'subscription',
      customer_email: req.user.email,
      line_items: [{ price: PLANS[plan].priceId, quantity: 1 }],
      success_url: `${req.headers.origin}/billing/success?session_id={CHECKOUT_SESSION_ID}`,
      cancel_url: `${req.headers.origin}/billing/cancel`,
    });
    res.json({ ok: true, url: session.url });
  } catch (e) {
    res.status(500).json({ ok: false, error: e.message });
  }
});
```

```
// Webhook — verify and update local subscription state
router.post('/webhook', require('express').raw({ type: 'application/json' }), (req, res) => {
  if (!stripe) return res.status(200).end();
  // Real impl: stripe.webhooks.constructEvent(req.body, sig, endpointSecret)
  // and switch on event.type to update Showroom.plan.
  res.status(200).end();
});

module.exports = router;
```

DEMO FILE back/bin/config.js

See how `stripeSecret` and `stripeWebhookSecret` read from env once, here. Every other file imports config — never `process.env` directly. This makes it trivial to mock for tests and impossible to leak a secret into a log.

```
// Central env loader. Every other module reads from here, never from process.env directly.
require('dotenv').config();

const config = {
  port: parseInt(process.env.PORT || '4000', 10),
  env: process.env.NODE_ENV || 'development',
  mongoUri: process.env.MONGODB_URI || 'mongodb://localhost:27017/auto-showroom',
  jwtSecret: process.env.JWT_SECRET || 'dev-secret-change-me',
  stripeSecret: process.env.STRIPE_SECRET_KEY || '',
  corsOrigin: process.env.CORS_ORIGIN || 'http://localhost:5173',
};

if (config.env === 'production' && config.jwtSecret === 'dev-secret-change-me') {
  console.warn('[config] JWT_SECRET is using the dev default in production. Set it in .env.');
}

module.exports = config;
```

CHAPTER 12 · THE FREE BUILDER COURSE

Growth Engine — Support, Marketing, Sales, Ops, And Not Burning Out

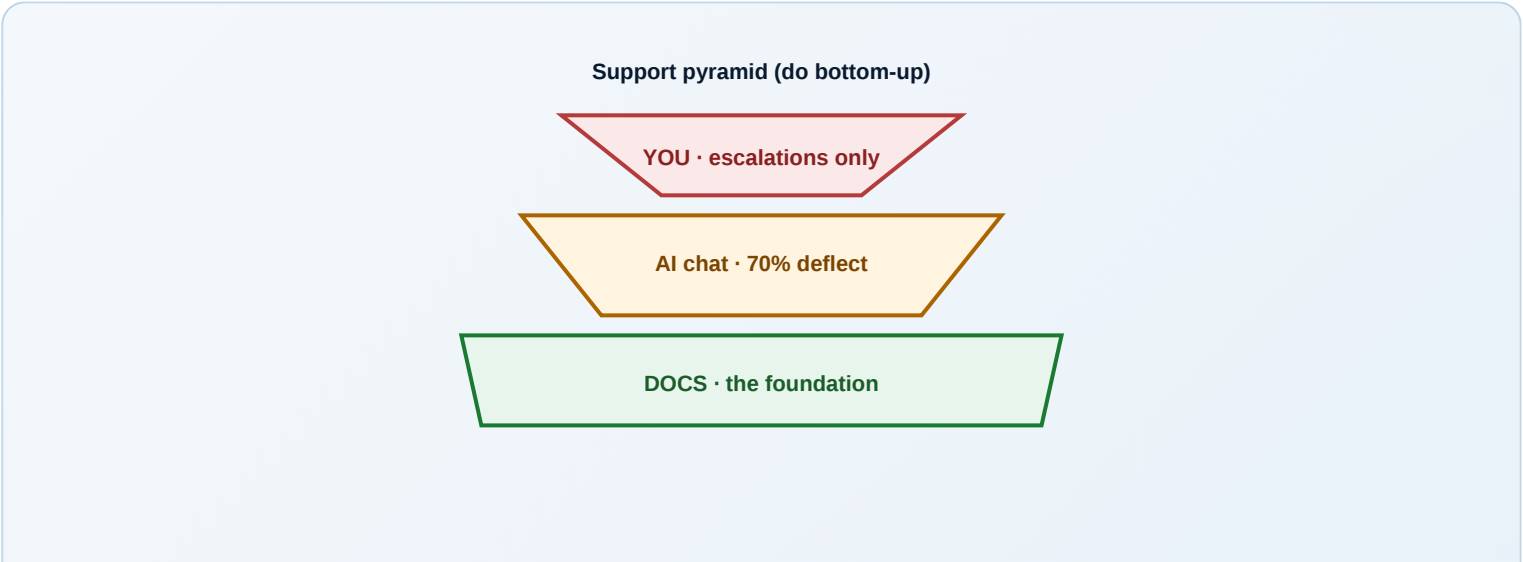
The flywheel that turns attention into revenue, with you in the loop only when it matters.

> *You have a product. You have customers. Now what? Now you build the engine that keeps strangers turning into customers while you sleep — and the parts that need you stay narrow enough that you actually do sleep.*

Support: docs, AI, human — in that order

Most solo SaaS founders get this exactly backwards. They start with "I'll answer every email personally", which works at 5 customers and explodes at 50. The order that scales is:

- **Docs first.** Every question you've answered twice belongs in the docs. Every question you've answered *once* belongs in the docs by tomorrow.
- **AI chat second.** Trained on your docs + transcripts. Handles 70% of new tickets without ever waking you.
- **Human (you) last.** Escalation only. The AI tags messages that need a human; you triage in a 20-minute morning batch.



Build the base wide. Most operators build it inverted, then drown.

Marketing: one idea, four formats

You don't have a marketing team. You have an AI worker and a Monday morning. Pick one idea per week and produce it in four formats:

- **Blog post** — long-form, SEO target query (the well from chapter 4)
- **Newsletter** — same idea, conversational, sent Friday
- **Video / short** — 90 seconds, the punchiest 1/3 of the post
- **Social posts** — the three sharpest sentences as standalone tweets / LinkedIn posts

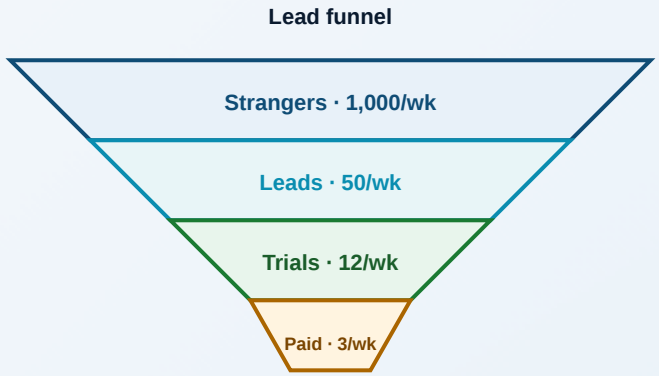
Same idea, four formats, four audiences, one Monday's work. Your AI worker drafts; you edit for voice. Ship Tuesday-Wednesday-Thursday-Friday.

> 🤖 **REAL WORKFLOW** I write the rough thesis Monday morning, paste it into Claude with "expand this into a 1,200-word post in my voice, here are five samples", spend 40 minutes editing, push. Tuesday: same thesis → "shorten to 300-word newsletter". Wednesday: same → "90-second video script". Thursday: same → "three Twitter posts, one LinkedIn". Total: one thesis, ~3 hours work, four channels.

Sales: serve first, sell second

Solo SaaS sales is not "cold calling." It's "showing up where your buyer already is, being useful, and being easy to buy from when they're ready." Three moves:

- **Lead capture, friction-free.** Public POST endpoint. No login required. Phone optional. Pre-fill what you can.
- **Funnel that nurtures, doesn't pester.** Welcome email + four-week drip + monthly newsletter. Unsub link in every one. The unsub link is a feature; it raises your sender score.
- **Closing is operator work.** You read every Enterprise demo request. You reply within 4 hours. You ask one question, not five. Sales is conversation; conversation is operator time.



The whole funnel on one diagram. Numbers are realistic for a $30K MRR SaaS in year two. Your numbers will differ; the shape will not.

Ops: every repeating task becomes code

Every Monday, list the things you did the previous week that you'd rather not do next week. Each one is a candidate for: a button in the admin page, a scheduled job, an AI-handled email rule, or a docs entry that prevents the question. After 12 weeks you have an admin cockpit so capable that you mostly just look at it.

- Recurring tickets → docs entry + AI chat coverage
- Recurring data fixes → admin button
- Recurring reports → scheduled Slack/email job
- Recurring onboarding handholding → in-product onboarding checklist

The part nobody writes about: not burning out

Burnout kills more solo SaaS than competition does. I will not lie to you: this is the part I am worst at. But I have rules and they help.

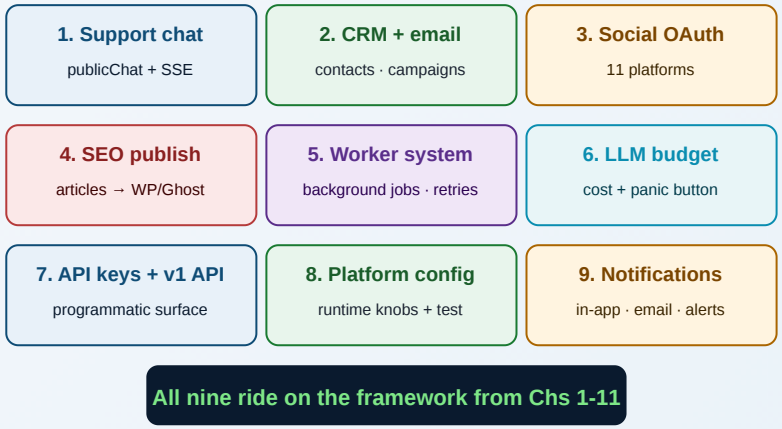
- **One full day a week off.** No code, no email, no Twitter. Sunday is mine.
- **Customer hours are bounded.** Email twice a day, not constantly. Slack notifications: off.
- **Ship something every Friday.** No matter how small. The shipping habit is the antidote to the "rebuilding from scratch" disease.
- **Talk to a human about the work.** Operator solitude is real. Join a community. (Free Builder is mine.)

▸ KEY TAKEAWAY

The growth engine is four loops: support (docs → AI → you), marketing (one idea, four formats), sales (serve, nurture, close), ops (every repeating task becomes code). The whole point of building all this is so you can step away from it on Sunday without anything falling over.

The nine sub-systems behind a real growth engine

The four loops above are the strategy. What follows is the **build**: nine concrete sub-systems that the Prigmar codebase ships and that any growth engine of size eventually needs. None of them is mandatory in v1. All of them earn their keep within twelve months. Treat this section as a roadmap.



Nine sub-systems. Each is its own slice of code. Each plugs into the auth, billing, and activity-log layers you've already built.

1. Customer-support chat (the public widget)

Prigmar ships a public-side support widget: an embed customers drop on their website, customers chat with your support bot, the conversation streams via Server-Sent Events back to the operator console. Five endpoints: `POST /chat`, `GET /chat`, `GET /publicChat/:widgetKey/conversations/:id`, `POST`

`/publicChat/:widgetKey/message`, `GET /publicChat/:widgetKey/conversations/:id/stream`. Models: `chatChannel`, `chatConversation`, `chatMessage`, `chatKnowledgeArticle`. The knowledge-article model is the secret: support questions get indexed into it, and the bot RAG-searches before escalating to you.

2. CRM + email (the contacts layer)

Once you have leads, you need a place to put them, a way to segment them, a way to email them. Prigmar's CRM ships `emailContact`, `emailContactList`, `emailMessage`, `emailTemplate`, `emailCampaign`, `emailIntent`. Endpoints: list contacts, get customers, get prospects, bulk update, AI-analyze a single contact (intent classification), update state/type. Newsletters: `newsletterIssue` + `newsletterSchedule`. Don't build this in v1; build it when your "growth engine" is bigger than your inbox can hold.

3. Social media OAuth (11 platforms)

If your product touches social media, you'll wire OAuth for at least three platforms; possibly eleven. The Prigmar pattern is one route per concern, with the platform name as a parameter: `GET /social/auth/:platform`, `GET /social/auth/:platform/callback`, `GET /social/:platform/auth-url`, `POST /social/:platform/connect-token`, `POST /social/:platform/refresh`, `POST /social/:platform/disconnect`. Plus account management: list connections, get account detail, get account targets, update target, cancel scheduled post. Models: `socialAccount`, `socialTarget`, `socialComment`, `socialAdsAccount`. The OAuth credentials per platform live in `back/bin/config.js` — extract them to env (see ch 3 graduation move #1) before someone audits the repo.

4. SEO publish pipeline

The companion to Ch 4. Prigmar's `SeoManagement` page is an in-app editor for SEO articles, briefs, and keywords. Models: `seoArticle`, `seoBrief`, `seoKeyword`, `seoFaq`, `seoPublishTarget`. The publish-target adapter is the killer feature: configure a target (WordPress site / Ghost blog / custom webhook), test it from

the admin UI, and pushes from in-app to your CMS happen with one click. Endpoints: SEO CRUD, plus `POST /seoPublishTargets/:id/test` for the "does my WordPress XML-RPC actually work?" sanity check.

5. Worker system (BullMQ-style)

You will outgrow doing work inside your HTTP handlers. Send-email-on-signup, generate-image-for-post, scrape-social-metrics — each is a candidate for a background job. Prigmar ships a worker system as a separate process: a pool manager that spawns N worker processes, each pulling jobs from an outbox-backed queue. Workers: `email-approval-worker`, `social-posting-worker`, `social-data-extractor-worker`, `text-generation-worker` (LLM calls), `token-refresh-worker`. A single proxy route lets the main app schedule jobs without holding a Redis connection directly. Graduation move #6 from Ch 3.

6. LLM budget + panic button

Already covered in Ch 9. Worth one bullet here as the operator's safety mechanism: `POST /llmBudget/panic` sets a hard cap that any LLM-touching middleware reads and short-circuits on. **Build this before your first viral moment.**

7. API keys + v1 contacts REST API

When customers ask "do you have a Zapier integration?" the honest answer is "we have an API; here are the keys." API-key CRUD is in Ch 6/7. The *customer-visible* REST endpoints come next: `POST /v1Contacts/contacts/upsert`, `GET /v1Contacts/contacts/:email`, `DELETE /v1Contacts/contacts/:email`, `POST /v1Contacts/contacts/event`, `GET /v1Contacts/contacts/export`. Five endpoints, versioned (`/v1Contacts/` not `/contacts/`) so you can iterate without breaking integrators.

8. Platform config (runtime-tunable knobs)

Some settings should be tunable without a redeploy: feature flags, kill switches, third-party API endpoints, rate limits. Prigmar's `platformConfig`: `GET /platformConfig` (all values), `PUT /platformConfig/:key` (update one), `POST /platformConfig/:key/test` (sanity check before committing), `DELETE /platformConfig/:key` (revert to default), `GET /platformConfig/sections` (group config by section for the admin UI). Backed by a single `platformConfig` model. The test endpoint is the underrated feature — every email setup screen needs a "send test" button.

9. Notifications + alerts

The companion to Ch 8's alert config. The notifications layer is the *delivery*: `notification` model (recipient, type, read status, metadata, payload), endpoints to list/create/mark-read. Combined with the alert config from Ch 8, you have a full notify pipeline: an event happens → alert config decides who cares → notification row written → email sent / in-app bell badged / Slack pinged. The activity log is the source; notifications are the sink.

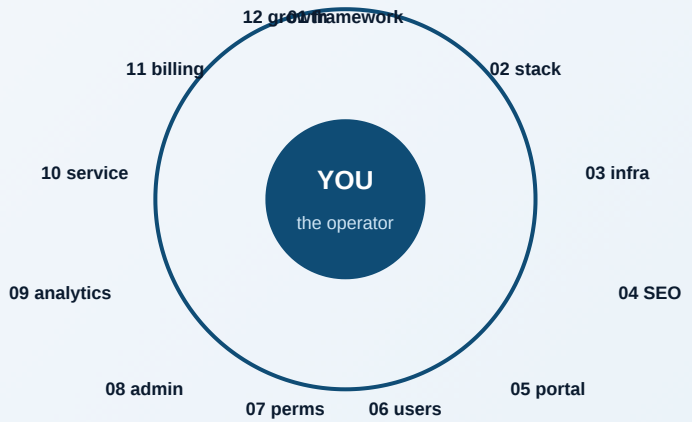
▸ ROADMAP, NOT A CHECKLIST

None of these nine sub-systems should ship in v1. Each is its own quarter's work. Add them in the order your customers (and your own pain) prioritize: usually support chat → CRM → notifications, then social/SEO/workers/LLM/API/config as the product matures. Use this section as the map for years 1 and 2.

And now you have a business

Fourteen days. Twelve chapters. One operator. If you've read every page, watched every slide, and read the code — congratulations, you are now in the very small minority of people who actually finish the thing they started.

The framework was never a secret. It was just twelve boring systems that compound when you do them in order. Two weeks to read; the rest of your operating life to apply. Every successful solo SaaS — mine, yours-in-progress, the next one I'll show you in v5 of this book — runs on the same shape. Different verticals, different verbs, same shape.



Twelve chapters around one operator. The circle is the framework. The circle is the moat. The circle is the business.

What's next

Three things. In this order.

1. **Open the demo** at `https://free-builder.com/cars/`. Click around. Read the code at `https://free-builder.com/cars/docs/`. Notice that it isn't magic — it's just the twelve systems wired together.
2. **Join the community** at skool.com/free-builder-8571. Get unstuck. Show your work. Find your operator friends.
3. **Ship something this week.** Not the perfect thing. Not the polished thing. *A* thing. Friday. Public. With a CTA.

> 🎬 **CLOSING** If you've read this far, I owe you one honest thing. Most courses end with "now you have everything you need." That's not quite true. You have the map. You still have to walk it. The walking is the part nobody can do for you, and the part that, when you do it, changes things. Go walk it.

▸ THE WHOLE FRAMEWORK, ONE SENTENCE

One operator, four pillars, twelve systems, an AI workforce — building businesses you own, one Friday at a time.

HELP THAT SCALES

Support Systems

Scalable support infrastructure

As your SaaS platform grows, user support can quickly become overwhelming. The solution? Implementing scalable, efficient support systems that work for you—even when you're offline.

Chatbots for instant assistance

A chatbot acts as your first line of defense in customer support. It's available around the clock, can answer FAQs, guide users through your app, or even escalate issues to your human team when necessary.

Contact forms for structured inquiries

For issues that require a human touch, contact forms allow users to submit their queries in a structured way. You'll gather the right info from the start, saving time for both you and the user.

Knowledge base for self-service

A searchable knowledge base or help center is essential. This is where you store guides, tutorials, FAQs, and troubleshooting content.

SEO, BLOG & FAQ

Content That Works While You Sleep

- SEO-driven content strategy
- Blog system as a long-term traffic generator
- FAQ system to reduce support tickets
- Content as an extension of your service
- Always-on value: content that works while you sleep

We don't just build software—we build ecosystems. Content isn't an afterthought. It's your silent sales rep and your most patient support agent.

SAAS MARKETING: GET PEOPLE IN

01

Content

- Blog posts targeting specific pain points
- Product explainers and use-case videos
- Free tools or calculators
- Lead magnets like eBooks, webinars, or email courses

02

Traffic

- How it works: Content fuels traffic through SEO, social sharing, email campaigns, and paid ads.
- Organic channels: Blogging, search engine optimization (SEO), social media, partnerships

03

Conversion

- Clear CTAs (calls-to-action)
- Landing pages optimized for pain points
- Free trials or freemium offers
- Case studies/testimonials for social proof

04

Retention

- Onboarding email flows
- In-app guidance and tooltips
- Continuous value delivery through updates
- Engagement through community or support

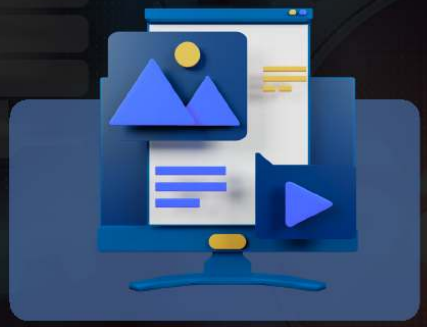
Marketing That Attracts & Converts

CHANNELS THAT WORK

Email, Content, Social, SEO

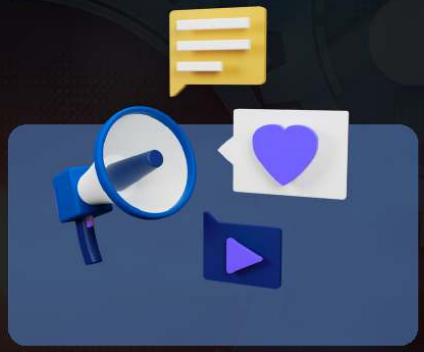
- **Email Marketing**

Email is still one of the highest ROI channels in SaaS. We'll break down how to write emails that don't just get opened—but actually convert. Whether it's onboarding sequences, product updates, or nurturing leads—every email you send should drive value and build trust



- **Content Marketing**

Content isn't just blog posts anymore. We'll show you how to create evergreen video content on platforms like YouTube that builds trust and educates your audience—bringing them closer to becoming paying users. You'll also learn how to structure blog content that ranks and resonates.



- **Social Media**

Social channels like Twitter can be goldmines for SaaS, especially in niche B2B spaces. Learn how to use organic posting, thought leadership, and direct engagement to build brand awareness, drive traffic, and even generate leads—without spending a dime on ads.

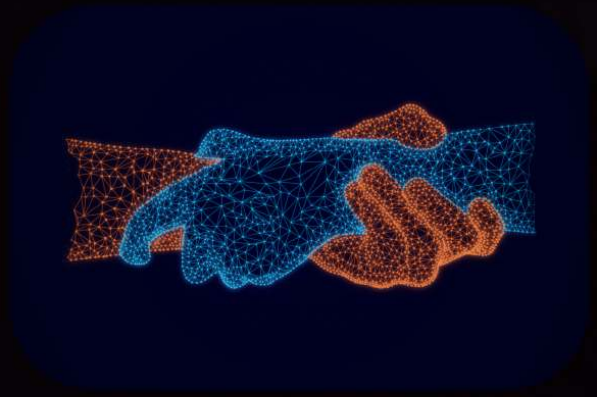
- **Search Engine Optimization (SEO)**

SEO is slow to start, but incredibly powerful. We'll cover how to optimize your content, target the right keywords, and structure your site to climb search rankings. The result? A steady stream of high-intent users discovering your product organically.

TEACH TO EARN TRUST

Content Strategy

- **Content = Scalable Trust-Building**
- **Educational Content Shows Authority & Understanding**
- **Attract Pre-Sold Users Through Value-Driven Education**
- **Content Types: Blogs, YouTube, Visual Tutorials**
- **Use a Consistent Content Calendar**



- Explain why your tool matters
- Solve real problems before a sales pitch
- Establish your team as experts in the niche

LEAD MAGNETS & EMAIL AUTOMATIONS

Capture > Nurture > Convert

Build systems to convert cold traffic into leads

- Lead Magnet Example: Offer a free resource such as a downloadable PDF guide, checklist, mini-course, or template related to your SaaS product's value.
- Tools: Use opt-in forms or popups on your site (via tools like ConvertKit, Mailchimp, or HubSpot).

Use email automation to build trust

- Email Sequence Should:
- Educate them about the problem your SaaS solves
- Share helpful content or use cases
- Showcase social proof (testimonials, case studies)
- Introduce product benefits gradually
- Include subtle CTAs (calls to action) leading to signup

Guide leads toward becoming paying customers

- Strategies to Convert:
- Offer a free trial or limited-time discount
- Use urgency or scarcity (e.g., “Only 3 spots left for onboarding this week!”)
- Offer 1-on-1 demos to answer questions live
- Include case studies showing real results

ORGANIC & PAID GROWTH

Own Your Traffic Engine

🌱 Organic Traffic: Build Sustainable Growth

SEO (Search Engine Optimization):

- Keyword Research: Learn how to identify high-intent keywords your target customers are searching for.
- On-Page SEO: Optimize your landing pages and blogs with clear titles, headers, and meta descriptions.
- Content Strategy: Create value-driven blog posts, tutorials, and landing pages around key problems your SaaS solves.
- Technical SEO: Ensure fast load times, mobile responsiveness, and proper indexing.

📢 Social Organic

- Share industry insights, customer wins, and micro-tutorials.
- Build a personal or brand voice to attract followers and community.
- Funnel attention back to landing pages or signup links.

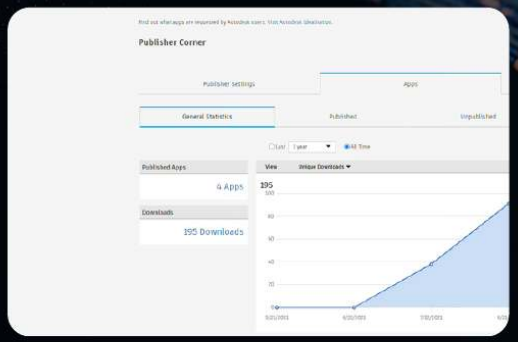
💰 Paid Traffic: Accelerate Your Results

- Google Ads (PPC Strategy):
- Campaign Setup: Create conversion-focused campaigns targeting keywords related to your product.
- Ad Copy: Craft headlines and descriptions that speak directly to pain points and benefits.
- Landing Page Alignment: Ensure the ad leads to a page that delivers on the promise and captures interest or signups.
- Tracking & ROI: Set up tracking via Google Analytics or your ad dashboard to measure performance and tweak over time.

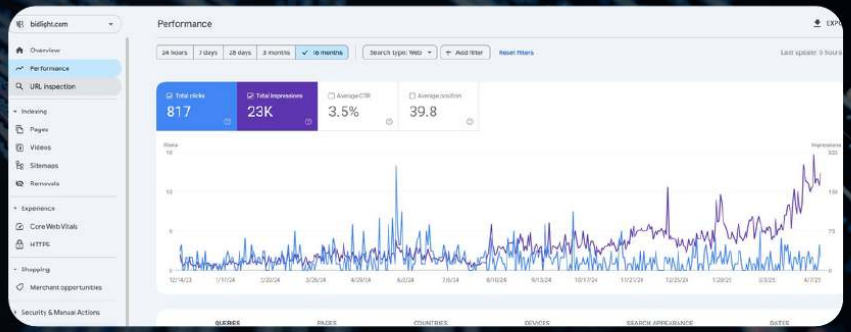
REAL GROWTH, REAL STORIES

How I Grew My SaaS (BidLight)

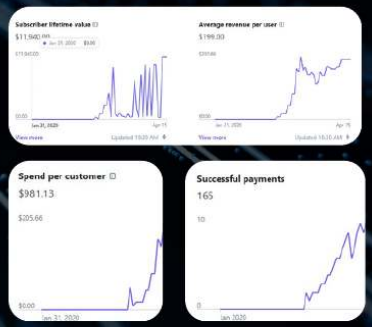
Custom Channel Growth



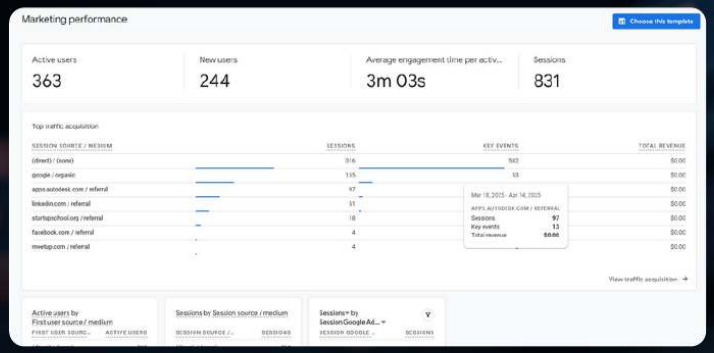
SEO Growth



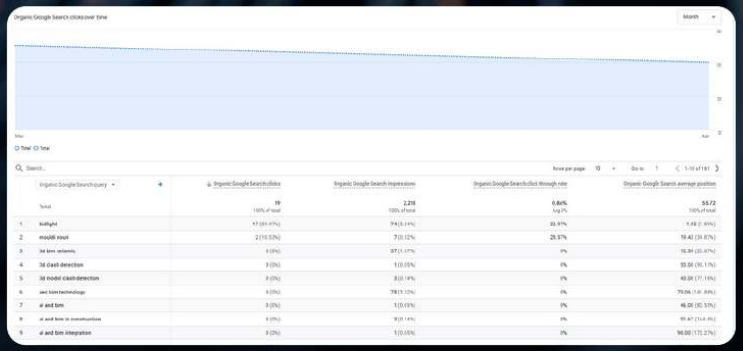
Stripe



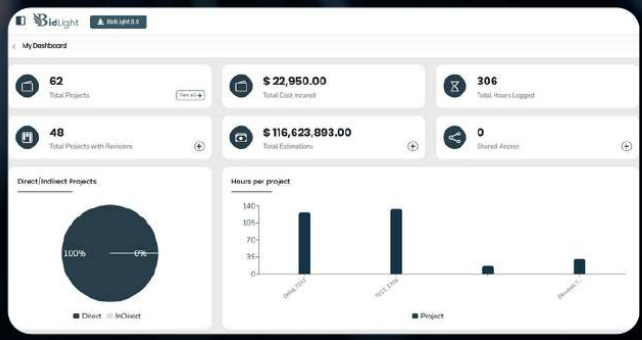
Marketing Performance



Organic Search



Analytics



TURN ATTENTION INTO REVENUE

Make Revenue Predictable

LEADS → FUNNEL → CLOSE → REVENUE

Leads: Users brought in by marketing efforts (ads, SEO, content, partnerships)

Funnel: The process by which leads are nurtured (emails, demos, webinars)

Close: The moment when a lead becomes a paying customer

Revenue: The result of optimized closing and retention

BUILD A SIMPLE FUNNEL

AWARENESS → EDUCATION → OFFER → CLOSE

Awareness

Introduce your product to potential users

Tools:

- Landing page with a clear headline
- Social media campaigns
- Google Ads or LinkedIn Ads

Education

Help users understand what your SaaS does and how it benefits them.

Tools:

- Email sequences (e.g., welcome series)
- Short explainer videos
- Educational blog posts or case studies

Offer

Present a solution and invite action..

Tools:

- Sales page or feature breakdown
- Personalized outreach (optional)
- Limited-time deals

Close

Convert the user into a paying customer

Tools:

- Testimonials, reviews, trust badges
- Live chat or consultation call
- Checkout flow / payment portal

SERVE FIRST, SELL SECOND

HELPING IS THE NEW SELLING.

- **Serve First, Sell Second**

In modern SaaS, hard selling is outdated. Instead of pushing products, focus on serving the user's needs first. Understand their challenges, and offer genuine solutions. Build trust before you pitch.

- **Helping is the New Selling**

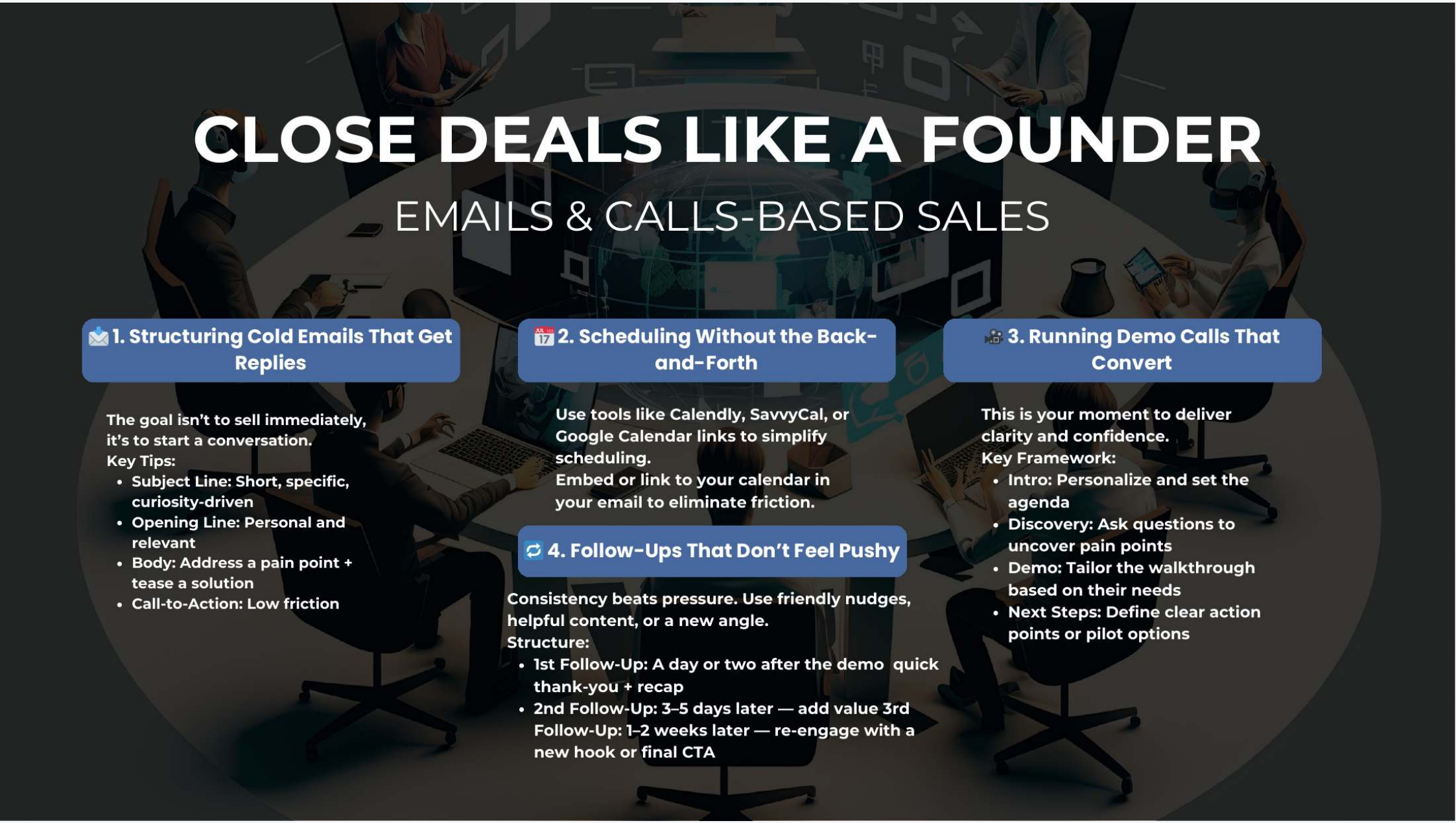
This quote sets the tone. True SaaS growth comes when your sales process feels like guidance, not pressure. Users don't want to be sold to — they want to be helped.

- **Educate and Align**

SaaS sales is essentially a teaching job. You're not just selling software — you're educating users on how your product can align with their goals.

- **Chase clarity**

If the product is a real match for the user, then you're not persuading — you're guiding. This flips the script from "salesperson" to "trusted advisor."



CLOSE DEALS LIKE A FOUNDER

EMAILS & CALLS-BASED SALES

📩 1. Structuring Cold Emails That Get Replies

The goal isn't to sell immediately, it's to start a conversation.
Key Tips:

- Subject Line: Short, specific, curiosity-driven
- Opening Line: Personal and relevant
- Body: Address a pain point + tease a solution
- Call-to-Action: Low friction

📅 2. Scheduling Without the Back-and-Forth

Use tools like Calendly, SavvyCal, or Google Calendar links to simplify scheduling.
Embed or link to your calendar in your email to eliminate friction.

🔁 4. Follow-Ups That Don't Feel Pushy

Consistency beats pressure. Use friendly nudges, helpful content, or a new angle.
Structure:

- 1st Follow-Up: A day or two after the demo quick thank-you + recap
- 2nd Follow-Up: 3–5 days later — add value 3rd Follow-Up: 1–2 weeks later — re-engage with a new hook or final CTA

🎥 3. Running Demo Calls That Convert

This is your moment to deliver clarity and confidence.
Key Framework:

- Intro: Personalize and set the agenda
- Discovery: Ask questions to uncover pain points
- Demo: Tailor the walkthrough based on their needs
- Next Steps: Define clear action points or pilot options

AFFILIATE & REFERRAL ENGINES

Turn Users Into Promoters

USER → UNIQUE LINK → NEW SIGNUP → REWARD TRIGGERED

User receives a personalized referral or affiliate link.

They share it via email, social media, or word-of-mouth.

A new user signs up using that link.

Both the referrer (and optionally the referee) get a reward.

SELL LIKE A PRO, NO MATTER THE MARKET

MAKE IT RUN WITHOUT YOU

The End Game – Systems Over Hustle

Product-market fit

- Customers genuinely need your product.
- Users are willing to pay and keep using it.
- Strong retention and engagement metrics.
- Feedback confirms you're solving the right pain point.

Customer onboarding

- Smooth, intuitive first-time experience.
- Helps users quickly understand and use core features.
- Drives user activation and early engagement.
- Reduces churn by preventing confusion or overwhelm.

Automated marketing

- Lead generation and nurturing run in the background.
- Personalized email sequences, retargeting, and CRM automations.
- Saves time and creates consistent user touchpoints.

Scalable support

- Support systems grow with user base.
- Mix of self-service (knowledge base) and live support.
- Reduces load on human agents through automation/chatbots.
- Enhances user satisfaction and loyalty.

Data-driven growth loops

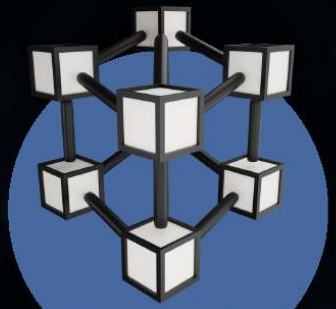
- Decisions based on metrics, not assumptions.
- Continuous feedback loops improve product and marketing.
- Growth systems become predictable and repeatable.

This module teaches you how to:

- **Automate repetitive processes**
- **Delegate operations using SOPs (Standard Operating Procedures)**
- **Integrate tools (like Zapier, HubSpot, Intercom) for smooth workflows**
- **Create a self-sustaining engine for growth**

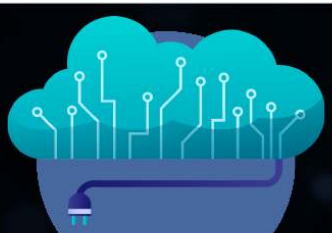
RECAP THE FLYWHEEL

STRUCTURE → SERVICE → MARKETING → SALES (REPEAT)



This is your foundation—your platform, features, and how your product is built.

- Focus on stability, scalability, and clarity
- Ensure it's easy to build upon and adapt
- A solid structure reduces friction in all other areas



Now you deliver value.

- Support, onboarding, and user education
- Responsive help and smart automation
- Building trust and solving real problems

TWith your product and service refined, now you attract attention.

- Share case studies, success stories
- Use content marketing, email campaigns, SEO
- Leverage social proof and testimonials

Good marketing brings interest, and good structure/service converts that interest into revenue.

- Streamline pricing models
- Offer demos, trials, or freemium
- Keep the pitch focused on value and simplicity

SYSTEMIZE ONBOARDING, BILLING, SUPPORT

The End Game – Systems Over Hustle

🧭 **Step 1: Signup**
📋 User signs up on your site or app
→ Trigger automation flow

📩 **Step 2: Welcome Email Sent**
💌 Personalized email delivered instantly
→ Includes login info, next steps, or a welcome video

🔓 **Step 3: Product Access Granted**
🔑 User gets immediate access to dashboard or app
→ Guided tour or checklist begins

🤖 **Step 4: Automated First-Level Support Enabled**
💬 Chatbot + Knowledge Base activated
→ Instant help for FAQs, setup issues, account details

DELEGATE WITH CONTRACTORS OR AI

YOU DON'T HAVE TO DO IT ALONE

- **Validate Your Systems**

Before you delegate, make sure your onboarding, support, billing, and marketing flows are working reliably.

- **Hire Contractors**

Once your system is stable, bring in freelance help to take on tasks that either drain your time or require skills outside your core strength.

- **Use AI and Automation Tools**

Technology can handle many repeatable tasks that used to require manual effort. Once you've identified areas where automation makes sense, implement AI or workflow automation tools.

- **Why This Matters**

When you delegate effectively—through people or tools—you unlock the ability to scale without burning out or bloating your team.


TURN CONTENT INTO COMPOUNDING TRAFFIC
BLOG → EMAIL → VIDEO → TWEET
START WITH A CORE PIECE OF CONTENT
CREATE A HIGH-VALUE, EVERGREEN BLOG POST:
• SOLVES A USER PAIN POINT
• SHOWS HOW YOUR PRODUCT HELPS
• EDUCATES WITH VISUALS, EXAMPLES, AND DATA
TURN IT INTO AN EMAIL NEWSLETTER
BREAK DOWN THE POST INTO A DIGESTIBLE EMAIL:
• USE THE INTRO OR KEY INSIGHT AS A HOOK
• LINK BACK TO THE FULL BLOG FOR DEEPER READING
• ADD A CTA TO TRY YOUR TOOL OR SIGN UP
CREATE A SHORT VIDEO OR SCREEN DEMO
REPURPOSE THE CONTENT VISUALLY:
• RECORD YOURSELF EXPLAINING THE BLOG'S KEY POINTS
• SHOW YOUR SAAS IN ACTION
• ADD CAPTIONS FOR SOCIAL AUTOPLAY
BREAK IT DOWN INTO MICRO CONTENT FOR SOCIAL
PULL QUOTES, STATS, OR TIPS FROM THE CONTENT:
• TWEET THREAD
• INSTAGRAM CAROUSEL
• LINKEDIN POST
THE MORE CONSISTENT YOU ARE, THE MORE YOUR PAST WORK KEEPS WORKING FOR YOU.

PROTECT YOUR ENERGY

AVOID FOUNDER BURNOUT

HUSTLE TO SYSTEM

- HUSTLE CULTURE LEADS TO BURNOUT
- SOLO SAAS NEEDS SYSTEMS, NOT NONSTOP GRIND
- LEVERAGE = WORKING SMARTER, NOT HARDER
- TIME PROTECTION IS YOUR SUPERPOWER
- MOMENTUM > BURNOUT

1. 💥 Hustle Isn't Sustainable
Nonstop work leads to exhaustion, not long-term success.

2. ⚙️ Build Systems, Not To-Do Lists
Automate and streamline repetitive tasks to free your mind.

3. 🛡️ Guard Your Time
Say no to distractions, batch your work, and prioritize deep focus.

4. 📈 Use Leverage
Use tools, automation, and contractors to get more done with less effort.

EXECUTE. SHARE. JOIN THE COMMUNITY.

FINAL WORDS

1. **Execute — Take action on what you've learned**
2. **Share Your Build — Show your progress and get visibility**
3. **Get Feedback — Improve faster with insights from others**
4. **Join the Community — Stay connected, inspired, and accountable**
5. **free-builder.com The central hub for connection**

FREE BUILDER

https://free-builder.com

THANKS

THE CODE, IN PRACTICE

From Auto Showroom — the car-sales reference codebase

The files below are the working implementation of this chapter's ideas. They live in the Auto Showroom demo at `https://free-builder.com/cars/` — anonymized from BidLight and Prigmar so you can see the same patterns without the proprietary details. Each file is annotated; read the commentary first, then the code.

DEMO FILE `back/routes/leads.js`

Three endpoints: public POST (anyone can leave a lead), staff GET (list), staff PUT (update status). The asymmetry is the whole point — the funnel *captures* publicly and *processes* privately. If your lead capture requires auth, you've already lost half your prospects.

```
const router = require('express').Router();
const ctrl = require('../controllers/lead.ctrl');
const requireAuth = require('../middlewares/requireAuth');
const requirePermission = require('../middlewares/requirePermission');

router.post('/', ctrl.create);  // public capture
router.get('/',  requireAuth, requirePermission(['sales','manager','admin']), ctrl.list);
router.put('/:id', requireAuth, requirePermission(['sales','manager','admin']), ctrl.updateStatus);

module.exports = router;
```

DEMO FILE back/controllers/lead.ctrl.js

Lead lifecycle: create from public form, list/filter for staff, update status with optional assignment, append notes. Read `updateStatus` end-to-end — the state transitions (`new` → `contacted` → `qualified` → `converted` / `lost`) ARE the sales process.

```
// Lead capture + management. Public can create; only staff can read/update.
const Lead = require('../models/Lead');
const activityLogs = require('../utils/activityLogs');

exports.create = async (req, res) => {
  const { name, email, phone, message, car, intent } = req.body || {};
  if (!name || !email) return res.status(400).json({ ok: false, error: 'Name and email required.' });
  const lead = await Lead.create({ name, email, phone, message, car, intent, source: 'web' });
  activityLogs.record({ action: 'lead.create', target: 'Lead', targetId: lead._id, req });
  res.status(201).json({ ok: true, lead });
};

exports.list = async (req, res) => {
  const { status, q, limit = 50, skip = 0 } = req.query;
  const filter = {};
  if (status) filter.status = status;
  if (q) filter.$or = [{ name: new RegExp(q, 'i') }, { email: new RegExp(q, 'i') }];
  const items = await Lead.find(filter).sort({ createdAt: -1 }).skip(Number(skip)).limit(Math.min(Number(limit),
200)).populate('car', 'make model year');
  const total = await Lead.countDocuments(filter);
  res.json({ ok: true, items, total });
};

exports.updateStatus = async (req, res) => {
  const lead = await Lead.findById(req.params.id);
  if (!lead) return res.status(404).json({ ok: false, error: 'Not found.' });
  const prev = lead.status;
  lead.status = req.body.status || lead.status;
  if (req.body.assignedTo) lead.assignedTo = req.body.assignedTo;
  if (req.body.note) lead.notes.push({ at: new Date(), by: req.user._id, text: req.body.note });
  await lead.save();
  activityLogs.record({ actor: req.user._id, action: 'lead.status_change', target: 'Lead', targetId: lead._id, diff: { from:
prev, to: lead.status }, req });
  res.json({ ok: true, lead });
};
```

DEMO FILE back/models/Lead.js

Lead schema. Notice `source`, `intent`, `status`, `assignedTo`, and a `notes` array of `{at, by, text}`. That's a one-table CRM. You don't need HubSpot for the first 200 leads; you need this schema.

```
// A lead is anyone who showed interest — test-drive booking, finance inquiry, plain contact.
const mongoose = require('mongoose');

const LeadSchema = new mongoose.Schema({
  showroom: { type: mongoose.Schema.Types.ObjectId, ref: 'Showroom', index: true },
  car:      { type: mongoose.Schema.Types.ObjectId, ref: 'Car', index: true },
  customer: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  name:     { type: String, required: true },
  email:    { type: String, required: true },
  phone:    String,
  message:  String,
  source:   { type: String, default: 'web' },   // web, phone, walk-in, referral, ad
  intent:   { type: String, enum: ['inquiry','test-drive','financing','offer'], default: 'inquiry' },
  status:   { type: String, enum: ['new','contacted','qualified','converted','lost'], default: 'new', index: true },
  assignedTo: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  notes:    [{ at: Date, by: mongoose.Schema.Types.ObjectId, text: String }],
  createdAt:{ type: Date, default: Date.now },
  updatedAt:{ type: Date, default: Date.now },
});

LeadSchema.pre('save', function (next) { this.updatedAt = new Date(); next(); });

module.exports = mongoose.model('Lead', LeadSchema);
```

You have the kit and the code. Now build something.

The framework only works when you put it to work. See the working version at free-builder.com/cars/. Read the module docs at free-builder.com/cars/docs/. Join the community to ship alongside others doing the same.

Join the community →