DABBAZ

Vibe Coding Guide with Cursor

Step-by-Step Build Guide

How to build Dabbaz from scratch using Cursor AI

Version 1.1 | February 2026

0. How to Use This Guide

This guide is your build companion. Keep it open in a separate window while you work in Cursor. It is structured as a strict sequence — each phase builds on the last. Do not skip ahead. The single biggest mistake in vibe coding a complex app is building UI before you have a solid data foundation underneath it.

Each section tells you exactly: what to do before opening Cursor, what to type into Cursor, and what to verify before moving on. The Cursor prompts are written to be copied almost verbatim — they are specific enough to get good output without being so long that the model loses track.

How this guide is structured

Phase 0: Project setup → Phase 1: Database schema → Phase 2: Auth → Phase 3: Core modules in order → Phase 4: Payments → Phase 5: Admin dashboards → Phase 6: Polish & deploy. Always complete a phase fully before starting the next.

1. Phase 0 — Before You Open Cursor

Do all of this first. These are decisions and setups that Cursor cannot make for you, and getting them wrong early means painful refactoring later.

1.1 Accounts to Create

Create accounts on all of the following services before writing a single line of code. You will need API keys from each one during setup.

Service	What It Does	URL
Microsoft Azure	Hosts your Node.js API (App Service), MySQL database, file storage (Blob), and Redis cache	portal.azure.com
Razorpay	Payments, subscriptions, vendor payouts	razorpay.com
Resend	Transactional email (receipts, notifications, OTPs)	resend.com
MSG91	SMS OTP for phone verification	msg91.com
Google Cloud Console	Google OAuth 2.0 credentials	console.cloud.google.com
Google reCAPTCHA	Anti-spam on vendor onboarding form	google.com/recaptcha
Firebase (FCM)	Push notifications to users	console.firebase.google.com

⚠ Watch Out

Do NOT use Razorpay live keys until you are ready to go live. Use test mode throughout all development. Mixing up test and live keys is a common and costly mistake.

1.2 Azure Resources to Provision

Log into portal.azure.com and create the following resources. Group them all under one Resource Group (e.g., 'dabbaz-rg') for easy management. Use the Free or Basic tier for everything during development.

- **Azure Database for MySQL — Flexible Server:** Your MySQL database. Note the server name, admin username, and password. Enable 'Allow public access from any Azure service' during dev.
- **Azure App Service:** This will host your Node.js backend API. Choose the Node 20 LTS runtime stack. Basic B1 tier is sufficient for MVP.
- **Azure Static Web Apps:** This will host your React frontend. Free tier is fine — it integrates directly with GitHub for auto-deploy.

- **Azure Blob Storage:** For vendor photos, menu images, and KYC documents. Create two containers: 'public-assets' (public read access) and 'private-docs' (private, SAS URL access only).
- **Azure Cache for Redis:** For Bull job queues and OTP rate limiting. Basic C0 tier during dev.

1.3 Install These on Your Machine

Open your terminal and run these before anything else:

```
# Node.js (use v20 LTS) — download from nodejs.org if needed
node --version   # Should print v20.x.x

# Install Cursor — download from cursor.com

# Install Azure CLI (for deployments later)
# Mac:    brew install azure-cli
# Windows: winget install Microsoft.AzureCLI
az --version

# MySQL Workbench — for viewing your database during dev
# Download from dev.mysql.com/downloads/workbench
```

1.4 Create Your Two Projects

You are building two separate codebases: a React frontend and a Node.js backend API. Create both before opening Cursor.

```
# 1. Create the React frontend (Vite)
npm create vite@latest dabbaz-frontend -- --template react
cd dabbaz-frontend
npm install
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p

# 2. Create the Node.js backend
mkdir dabbaz-backend && cd dabbaz-backend
npm init -y
npm install express cors helmet dotenv express-validator
npm install passport passport-google-oauth20 passport-jwt jsonwebtoken bcrypt
npm install prisma @prisma/client
npm install razorpay resend bull ioredis
npm install @azure/storage-blob
npm install -D nodemon typescript ts-node @types/node @types/express
npm install -D @types/passport @types/jsonwebtoken @types/bcrypt

# 3. Frontend additional packages
cd ../dabbaz-frontend
npm install react-router-dom axios react-hook-form zod @hookform/resolvers
npm install @tanstack/react-query date-fns lucide-react clsx
```

Pro Tip

Open both projects in Cursor at the same time by opening the parent folder that contains both: cursor . (from the folder above dabbaz-frontend and dabbaz-backend). You can then navigate between both in the file tree.

1.5 Set Up Your Environment Files

Create a .env file in the backend and a .env file in the frontend. Never commit these to Git — add them to .gitignore immediately.

Backend — dabbaz-backend/.env:

```
# Database
DATABASE_URL=mysql://adminuser:password@your-azure-mysql-
server.mysql.database.azure.com:3306/dabbaz

# JWT
JWT_SECRET=a_very_long_random_string_at_least_64_chars
JWT_REFRESH_SECRET=another_very_long_random_string
JWT_EXPIRES_IN=15m
JWT_REFRESH_EXPIRES_IN=30d

# Google OAuth
GOOGLE_CLIENT_ID=your_google_client_id
GOOGLE_CLIENT_SECRET=your_google_client_secret
GOOGLE_CALLBACK_URL=http://localhost:4000/api/auth/google/callback

# Razorpay (TEST keys during dev)
RAZORPAY_KEY_ID=rzp_test_xxxxxxxxxxxx
RAZORPAY_KEY_SECRET=your_razorpay_secret

# MSG91
MSG91_AUTH_KEY=your_msg91_key
MSG91_TEMPLATE_ID=your_otp_template_id

# Resend
RESEND_API_KEY=re_xxxxxxxxxxxx

# Azure Blob Storage
AZURE_STORAGE_CONNECTION_STRING=DefaultEndpointsProtocol=https;...
AZURE_STORAGE_PUBLIC_CONTAINER=public-assets
AZURE_STORAGE_PRIVATE_CONTAINER=private-docs

# Redis (Azure Cache for Redis)
REDIS_URL=rediss://your-redis.redis.cache.windows.net:6380
REDIS_PASSWORD=your_redis_access_key

# reCAPTCHA
RECAPTCHA_SECRET_KEY=your_secret_key

# App
PORT=4000
FRONTEND_URL=http://localhost:5173
PLATFORM_COMMISSION_RATE=0.12
```

Frontend — dabbaz-frontend/.env:

```
VITE_API_BASE_URL=http://localhost:4000/api
VITE_RAZORPAY_KEY_ID=rzp_test_xxxxxxxxxxxx
VITE_RECAPTCHA_SITE_KEY=your_recaptcha_site_key
```

2. Phase 1 — Database Schema (Do This Before Any UI)

This is the most important phase. The schema is the skeleton of your entire app. Every feature you build later will read from and write to these tables. If the schema is wrong, everything built on top of it will eventually need to be refactored. This is not the place to let Cursor improvise.

Pro Tip

Read the entire schema section before giving anything to Cursor. Understand what each table does and why. Cursor will ask clarifying questions — you need to be able to answer them.

2.1 Initialize Prisma in the Backend

In your terminal, navigate to your backend folder:

```
cd dabbaz-backend
npx prisma init
```

This creates a prisma/schema.prisma file. Open it in Cursor and delete everything inside it. Also open prisma/.env and set your DATABASE_URL to your Azure MySQL connection string.

⚠ Watch Out

MySQL does not support native array column types. Fields like cuisine_tags, delivery_pincodes, and photo_urls must be stored as Json type in Prisma (maps to JSON column in MySQL). When querying these in your API, parse them with JSON.parse() after reading and JSON.stringify() before writing. Do NOT use String[] in the schema — Prisma will error on MySQL.

2.2 The Complete Schema — Copy This Into Cursor

Open Cursor chat (CMD+L) with the backend folder open, and paste this prompt:

```
I'm building Dabbaz, a tiffin subscription marketplace. I'm in the
dabbaz-backend folder. Create a complete Prisma schema in
prisma/schema.prisma for the following tables.
Use MySQL (datasource provider = 'mysql').
```

```
Do not use String[] arrays — MySQL does not support them.
Use Json type for array-like fields (cuisine_tags, delivery_pincodes,
photo_urls, tags). Do not invent extra tables or fields.
After writing the schema, write the migration command to run.

Tables and fields:

User: id (Int autoincrement), email (unique), name, phone (unique?),
  phone_verified (Boolean default false), avatar_url, role
  (CUSTOMER | VENDOR | ADMIN), wallet_balance (Decimal default 0),
  referral_code (unique), referred_by_id (Int?, self-relation to User),
  created_at, updated_at

VendorProfile: id, user_id (User, unique), business_name,
  slug (unique), about, fssai_number, fssai_doc_url, govt_id_url,
  is_verified (Boolean default false), is_active (Boolean default true),
  cuisine_tags (Json), food_type (VEG | NONVEG | BOTH),
  lunch_window_start (String?), lunch_window_end (String?),
  dinner_window_start (String?), dinner_window_end (String?),
  delivery_pincodes (Json), cover_photo_url, photo_urls (Json),
  commission_rate (Decimal default 0.12),
  daily_capacity (Int default 999),
  active_subscriber_count (Int default 0),
  bank_account_name, bank_account_number, bank_ifsc,
  created_at, updated_at

VendorOnboardingRequest: id, user_id (User), status
  (PENDING | APPROVED | REJECTED | NEEDS_MORE_INFO), business_name,
  contact_name, address, pincode, years_of_operation (Int),
  daily_capacity (Int), fssai_doc_url, govt_id_url,
  hygiene_cert_url, sample_menu_text (Text), rejection_reason,
  admin_notes, recaptcha_score (Float), created_at, updated_at

SubscriptionPlan: id, vendor_id (VendorProfile), name, description,
  duration_days (Int, min 4), meal_type (LUNCH | DINNER | BOTH),
  food_type (VEG | NONVEG | BOTH), price (Decimal),
  is_active (Boolean default true), auto_renewal_default (Boolean),
  created_at, updated_at

MenuItem: id, vendor_id (VendorProfile), date (DateTime),
  meal_type (LUNCH | DINNER), name, description (Text?),
  food_type (VEG | NONVEG), photo_url,
  is_off_day (Boolean default false),
  is_slot_disabled (Boolean default false),
  is_published (Boolean default false), created_at, updated_at
  @@unique([vendor_id, date, meal_type])

Addon: id, vendor_id (VendorProfile), name, price (Decimal),
  food_type (VEG | NONVEG), is_active (Boolean default true),
  created_at

MenuItemAddon: id, menu_item_id (MenuItem), addon_id (Addon)
  @@unique([menu_item_id, addon_id])

Subscription: id, user_id (User), plan_id (SubscriptionPlan),
  vendor_id (VendorProfile), status (ACTIVE | PAUSED | CANCELLED |
  EXPIRED), start_date (DateTime), end_date (DateTime),
  auto_renewal (Boolean), delivery_notes (Text?),
  dietary_preference (String?), meals_remaining (Int),
  resume_date (DateTime?), created_at, updated_at
```

```
Order: id, subscription_id (Int?, FK to Subscription),
  user_id (User), vendor_id (VendorProfile),
  delivery_date (DateTime), meal_type (LUNCH | DINNER),
  status (PREPARING | OUT_FOR_DELIVERY | DELIVERED | NOT_DELIVERED),
  is_trial (Boolean default false), created_at, updated_at

OrderAddon: id, order_id (Order), addon_id (Addon), quantity (Int)

Payment: id, user_id (User), order_id (Int?), subscription_id (Int?),
  razorpay_order_id (String unique), razorpay_payment_id (String?),
  amount (Decimal), platform_fee (Decimal), vendor_payout (Decimal),
  status (PENDING | SUCCESS | FAILED | REFUNDED), created_at

Dispute: id, user_id (User), order_id (Order), vendor_id (VendorProfile),
  type (NON_DELIVERY | WRONG_ITEM | QUALITY | OVERCHARGE),
  description (Text), photo_url, status
  (OPEN | VENDOR_RESPONDED | RESOLVED | DISMISSED),
  vendor_response (Text?), admin_resolution (Text?),
  credit_issued (Decimal?), created_at, updated_at

WalletTransaction: id, user_id (User), amount (Decimal), type
  (CREDIT | DEBIT), reason (String), reference_id (String?), created_at

VendorPayout: id, vendor_id (VendorProfile), amount (Decimal),
  status (PENDING | PROCESSED | FAILED), period_start (DateTime),
  period_end (DateTime), processed_at (DateTime?),
  utr_number (String?), created_at

Review: id, user_id (User), vendor_id (VendorProfile),
  rating (Int), comment (Text), tags (Json),
  photo_url, vendor_response (Text?), is_flagged (Boolean default false),
  is_hidden (Boolean default false), created_at, updated_at
  @@unique([user_id, vendor_id])

Notification: id, user_id (User), title, body (Text), type (String),
  is_read (Boolean default false), meta (Json?), created_at

RefreshToken: id, user_id (User), token (String unique),
  expires_at (DateTime), created_at
```

⚠ Watch Out

After Cursor writes the schema, read through every table before running the migration. Check that no String[] arrays snuck in, all Decimal fields have the right precision, and the @@unique constraints are present. Ask Cursor to fix anything before proceeding.

After Cursor Writes the Schema, Run:

```
cd dabbaz-backend
npx prisma migrate dev --name init
npx prisma generate
```

If the migration fails, paste the full error back into Cursor. Do not edit migration files manually.

2.3 Verify in MySQL Workbench

Open MySQL Workbench, connect to your Azure MySQL server, and run:

```
USE dabbaz;
SHOW TABLES;
```

You should see all tables listed. If any are missing, check the terminal output from the migration command for errors.

3. Phase 2 — Project Structure & Folder Layout

You have two separate projects. Give both a clean, predictable folder structure before writing any feature code. Cursor works much better when files are in consistent, logical locations. Run these two prompts — one per project.

3.1 Backend Structure

```
I'm in the dabbaz-backend folder. Set up the following folder structure.
Create each folder and add a .gitkeep file inside empty ones.
Do not write any code yet — just the folders and a basic Express app
entry point at src/index.ts that listens on port 4000.

src/
  index.ts              ← entry point
  app.ts                ← Express setup (cors, helmet, json middleware)
  routes/
    auth.routes.ts
    vendor.routes.ts
    menu.routes.ts
    subscription.routes.ts
    payment.routes.ts
    order.routes.ts
    dispute.routes.ts
    admin.routes.ts
    webhook.routes.ts
  controllers/
    auth.controller.ts
    vendor.controller.ts
    menu.controller.ts
    subscription.controller.ts
    payment.controller.ts
    order.controller.ts
    dispute.controller.ts
    admin.controller.ts
  middleware/
    auth.middleware.ts    ← JWT verification, attaches user to req
    role.middleware.ts    ← requireRole('VENDOR'), requireRole('ADMIN')
    upload.middleware.ts  ← Azure Blob upload handler
    validate.middleware.ts
  lib/
    prisma.ts             ← Prisma client singleton
    razorpay.ts
    resend.ts
    azure-storage.ts      ← Blob upload/SAS URL helpers
    redis.ts              ← Redis + Bull queue setup
    passport.ts           ← Passport.js strategy config
  jobs/
    subscription.job.ts
    notification.job.ts
    payout.job.ts
  types/
    express.d.ts          ← extend Express Request (add req.user)
    index.ts
  utils/
    jwt.ts
    crypto.ts
    otp.ts
```

3.2 Frontend Structure

```
I'm in the dabbaz-frontend folder. Set up the following folder structure
inside src/. Create each folder and add a .gitkeep inside empty ones.
Do not write any component code — just the folders. Also update
tailwind.config.js to scan all files in src/.

src/
  main.tsx
  App.tsx                ← React Router v6 routes
  pages/
    public/              ← No auth required
      HomePage.tsx
      VendorProfilePage.tsx
    auth/
      LoginPage.tsx
      SignupPage.tsx
    customer/
      DashboardPage.tsx
      SubscriptionsPage.tsx
      CheckoutPage.tsx
      PaymentsPage.tsx
      ProfilePage.tsx
      BecomeVendorPage.tsx
    vendor/
      VendorDashboardPage.tsx
      MenuPage.tsx
      OrdersPage.tsx
      EarningsPage.tsx
      SubscribersPage.tsx
      PlansPage.tsx
      VendorSettingsPage.tsx
    admin/
      AdminDashboardPage.tsx
      VendorQueuePage.tsx
      UsersPage.tsx
      DisputesPage.tsx
      PayoutsPage.tsx
  components/
    ui/
    vendor/
    customer/
    admin/
    layout/
  hooks/
  lib/
    api.ts              ← Axios instance (base URL + auth header)
    queryClient.ts      ← React Query setup
  store/
    auth.store.ts       ← Auth context / useAuth hook
  types/
    index.ts
  constants/
    index.ts
  utils/
    index.ts
```

4. Phase 3 — Authentication

Auth is the foundation everything else depends on. You are building auth from scratch using Passport.js for Google OAuth and JWTs for session management. Build this completely — and test it — before touching any other feature.

4.1 Configure Google OAuth

In Google Cloud Console:

1. Create a new project (or use an existing one).
2. Go to APIs & Services → Credentials → Create OAuth 2.0 Client ID.
3. Application type: Web application.
4. Add http://localhost:4000/api/auth/google/callback to Authorised Redirect URIs.
5. Copy the Client ID and Client Secret into your backend .env file.

4.2 Build the Backend Auth System

```
I'm in dabbaz-backend. Build the complete auth system.

1. src/lib/passport.ts — Configure Passport.js with two strategies:

   a) GoogleStrategy (passport-google-oauth20):
      - Callback URL: process.env.GOOGLE_CALLBACK_URL
      - On success: find or create a User record in Prisma by email.
        If creating: set role=CUSTOMER, wallet_balance=0,
        generate a unique 8-char alphanumeric referral_code.
      - Return the User record as the passport profile.

   b) JwtStrategy (passport-jwt):
      - Extract JWT from Authorization header as Bearer token.
      - Verify using JWT_SECRET from env.
      - On success: fetch the User from Prisma by id in the JWT payload
        and attach to req.user.
```

```
2. src/utils/jwt.ts — Two functions:
   - generateAccessToken(userId): signs a JWT with 15min expiry
   - generateRefreshToken(userId): signs a JWT with 30d expiry,
     saves it to the RefreshToken table in the database

3. src/routes/auth.routes.ts — Routes:
   GET  /api/auth/google              → passport.authenticate('google', scopes:
email + profile)
   GET  /api/auth/google/callback     → on success, generate access + refresh
tokens,
                                        redirect to frontend with tokens as query
params
   POST /api/auth/register            → email/password signup
   POST /api/auth/login               → email/password login
   POST /api/auth/refresh             → accept refresh token, return new access
token
   POST /api/auth/logout              → delete RefreshToken from DB
   POST /api/auth/send-otp            → send SMS OTP via MSG91
   POST /api/auth/verify-otp          → verify OTP with MSG91,
                                        update user.phone + phone_verified

4. src/controllers/auth.controller.ts — Implement all above routes.
   For register: hash password with bcrypt (rounds: 12) before saving.
   For login: compare with bcrypt.compare(), reject if no match.
   Never return the password field in any response.

5. src/middleware/auth.middleware.ts
   - requireAuth: passport.authenticate('jwt', { session: false })
   - Attach req.user = the full User record from Prisma

6. src/middleware/role.middleware.ts
   - requireRole(...roles): middleware that checks req.user.role
     against allowed roles, returns 403 if not permitted
```

4.3 Build the Frontend Auth

Build the auth store and pages. Do login and signup as separate prompts.

```
I'm in dabbaz-frontend. Build the auth layer.

1. src/lib/api.ts — Axios instance:
   - baseURL: import.meta.env.VITE_API_BASE_URL
   - Request interceptor: attach access token from localStorage
     to every request as 'Authorization: Bearer <token>'
   - Response interceptor: if 401 received, call POST /api/auth/refresh
     with the refresh token from localStorage, update stored access token,
     and retry the original request once. If refresh fails, clear tokens
     and redirect to /login.

2. src/store/auth.store.ts — useAuth hook using React Context:
   - State: user (User object | null), isLoading
   - Actions: login(email, password), logout(), setUser(user)
   - On mount: if access token exists in localStorage,
     call GET /api/auth/me to hydrate the user object.

3. src/App.tsx — React Router v6 setup with protected routes:
   - ProtectedRoute component that reads from useAuth()
   - Redirects to /login if not authenticated
   - VendorRoute that also checks role === 'VENDOR'
   - AdminRoute that also checks role === 'ADMIN'
```

```
I'm in dabbaz-frontend. Build the Login page at src/pages/auth/LoginPage.tsx.

- Email + password form with react-hook-form and zod validation
- On submit: POST to /api/auth/login via the api.ts axios instance
  Store access_token and refresh_token in localStorage.
  Call setUser() from useAuth() with the returned user object.
  Redirect to /dashboard.
- 'Sign in with Google' button: opens window.location.href to
  import.meta.env.VITE_API_BASE_URL + '/auth/google'
  After redirect back, parse tokens from URL query params,
  store in localStorage, and redirect to /dashboard.
- 'Forgot password' link (placeholder for now)
- Link to /signup
- Inline field-level error messages
- Tailwind styling, mobile-first, no UI library
```

```
I'm in dabbaz-frontend. Build the Signup page at src/pages/auth/SignupPage.tsx.

- Fields: Full Name, Email, Password (min 8 chars), Confirm Password
- react-hook-form + zod validation
- On submit: POST to /api/auth/register
  On success, auto-login with the returned tokens (same as login flow)
  and redirect to /dashboard.
- 'Sign up with Google' button — same as login Google button
- Link to /login
- Tailwind styling, mobile-first
```

5. Phase 4 — Building Features in the Right Order

Now you start building actual features. Follow this order strictly. Each phase depends on the previous one being complete and tested.

#	Module	Why This Order
1	Vendor Onboarding Request	Needed before any vendor data exists. Sets up the approval queue.
2	Platform Admin Dashboard (basic)	You need to be able to approve vendors before vendors can do anything.
3	Vendor Profile Setup	Approved vendors need to configure their profile before listing.
4	Menu Builder	Vendors need a menu before users can subscribe.
5	Subscription Plans	Plans are created by vendors and purchased by users.
6	Public Discovery Page	Users can now browse real vendor data.

7	Vendor Public Profile Page	Full vendor storefront with menu and plans.
8	Payments & Subscriptions	Users can now subscribe and pay.
9	Customer Dashboard	Users manage their active subscriptions.
10	Delivery Status System	Vendors update delivery status; users see it.
11	Notifications	Email + push for all key events.
12	Disputes & Support	Users report issues; admin resolves.
13	Ratings & Reviews	Trust layer on top of a working platform.
14	Vendor Payout Dashboard	Revenue visibility for vendors.

6. Cursor Prompts — Module by Module

Below are the exact Cursor prompts to use for each module. Each prompt is designed to be focused and specific. Avoid the temptation to combine multiple modules into one prompt — Cursor's output quality drops significantly with overly broad requests.

Module 1 — Vendor Onboarding Request

```
Build the Vendor Onboarding Request flow.

1. Create a page at src/app/(customer)/become-a-vendor/page.tsx
   This is a multi-step form (4 steps shown as a progress indicator at top):

   Step 1 — Phone Verification
   - Input field for Indian mobile number (10 digits)
   - 'Send OTP' button that calls POST /api/auth/send-otp
   - OTP input (6 digits) that appears after sending
   - 'Verify' button that calls POST /api/auth/verify-otp
   - On success, mark phone as verified in state and proceed to Step 2

   Step 2 — Business Details
   - Fields: Business Name, Contact Person, Full Address, PIN Code,
     Years of Operation (number), Daily Capacity (number of tiffins)
   - react-hook-form + zod validation

   Step 3 — Documents
   - File upload for FSSAI License (PDF/JPG, max 5MB) — required
   - File upload for Government ID (PDF/JPG, max 5MB) — required
   - File upload for Hygiene Certificate (PDF/JPG, max 5MB) — optional
   - On file select, call POST /api/upload/private-doc (multipart/form-data).
     The backend uploads to Azure Blob private container and returns a
     document_url. Store this URL in form state.

   Step 4 — Sample Menu & Submit
   - Textarea: describe typical meals you offer (min 100 chars)
   - Google reCAPTCHA v3 widget (invisible, fires on submit)
   - Declaration checkbox: 'I confirm all information is accurate'
   - Submit button calls POST /api/vendors/onboarding-request

2. Create API route POST /api/auth/send-otp
   - Validate phone number format
   - Check rate limit: max 3 OTP sends per phone per hour (use a simple
     in-memory store or Redis if available)
   - Call MSG91 API to send OTP
   - Return success/error

3. Create API route POST /api/auth/verify-otp
   - Verify OTP with MSG91
   - On success, update User.phone and User.phone_verified in database
   - Return success/error

4. Create API route POST /api/vendors/onboarding-request
   - Require authenticated user
   - Require phone_verified = true on user record
   - Verify reCAPTCHA token server-side (reject if score < 0.5)
   - Check for duplicate: same phone or same FSSAI number already in
     VendorOnboardingRequest table → return 409 error
   - Create VendorOnboardingRequest record with status PENDING
     and store recaptcha_score
   - Send confirmation email via Resend to user
   - Return success
```

Module 2 — Admin Dashboard (Basic — Vendor Approval)

```
Build the basic Admin Dashboard focused on vendor approval.

1. Create layout at src/app/(admin)/layout.tsx
   - Sidebar navigation with links to: Overview, Vendor Queue,
     Vendors, Users, Disputes, Payouts
   - Only accessible to users with role = ADMIN
   - If not admin, redirect to /dashboard

2. Create page src/app/(admin)/admin-dashboard/page.tsx
   - 4 stat cards: Total Users, Active Vendors, Pending Applications,
     Active Subscriptions
   - Fetch counts via GET /api/admin/stats

3. Create page src/app/(admin)/vendors/queue/page.tsx
   - Table of all VendorOnboardingRequests with status PENDING
   - Columns: Submitted date, Business Name, Contact, PIN Code,
     reCAPTCHA Score, Status, Actions
   - Clicking a row opens a slide-over panel showing all details
     including document links (SAS URLs from Azure Blob private container,
     generated server-side via GET /api/admin/vendor-queue/:id/doc-url?file=fssai)
   - Three action buttons: Approve, Reject (with reason modal),
     Request More Info (with message modal)

4. Create API route GET /api/admin/stats — returns user/vendor/subscription counts

5. Create API route GET /api/admin/vendor-queue — returns pending applications

6. Create API route POST /api/admin/vendor-queue/[id]/approve
   - Require ADMIN role
   - Update VendorOnboardingRequest status to APPROVED
   - Create VendorProfile record for the user
   - Update User.role to VENDOR
   - Send welcome email to vendor via Resend

7. Create API route POST /api/admin/vendor-queue/[id]/reject
   - Require ADMIN role
   - Accept rejection_reason in body
   - Update status to REJECTED, store reason
   - Send rejection email with reason to applicant

8. Create API route POST /api/admin/vendor-queue/[id]/request-info
   - Update status to NEEDS_MORE_INFO
   - Send email with the admin's message to applicant
```

Module 3 — Vendor Profile Setup

```
Build the Vendor Profile Settings page.

Create page at src/app/(vendor)/settings/page.tsx
This page lets vendors configure their public profile.

Sections (use tabs or accordion):

1. Basic Info
   - Business name (pre-filled from onboarding), about (rich text — use
     a simple textarea, not a rich text editor), cuisine tags
     (multi-select from predefined list), food type (Veg / Non-Veg / Both)

2. Delivery Settings
   - Delivery PIN codes: tag-input where vendor types a PIN and presses
     Enter to add it. Show added PINs as removable chips.
   - Lunch delivery window: two time inputs (from / to)
   - Dinner delivery window: two time inputs (from / to)

3. Photos
   - Upload up to 8 photos via POST /api/upload/public-asset (multipart/form-data).
     Backend uploads to Azure Blob public container and returns a photo_url.
   - Show existing photos in a grid with a delete button on each
   - First photo becomes cover. Allow drag to reorder.

4. Bank Details (for payouts)
   - Bank account holder name, account number (masked after save),
     IFSC code
   - Show a disclaimer: 'Your bank details are encrypted and used only
     for payouts.'

5. Capacity & Availability
   - Daily Capacity: number input labelled 'Max subscribers per day'.
     Show current active subscriber count next to it so vendor knows
     their headroom (e.g., '18 of 25 slots filled').
     Saving this updates VendorProfile.daily_capacity.
     If vendor reduces capacity below current active_subscriber_count,
     show a warning but allow it — new subscribers will simply be blocked
     until cancellations bring the count below the new cap.
   - Availability Toggle: 'Accepting new subscriptions' toggle switch.
     When toggled off, show a confirmation dialog explaining that
     existing subscribers are not affected. This sets is_active = false
     on the VendorProfile (separate from capacity — this is a manual override).
```

```
All sections save independently via PATCH /api/vendors/profile
Show a toast notification on save success/failure.
```

Module 4 — Menu Builder

This is the most complex UI in the project. Build it in two sub-steps.

```
STEP 4A — Build the Menu Builder page at src/app/(vendor)/menu/page.tsx

Layout:
- Week navigator at the top: left/right arrows to change week,
  showing 'Mon 10 Feb — Sun 16 Feb' format. Can navigate up to 8 weeks ahead.
- Two tabs: Lunch | Dinner
- A 7-column grid (Mon to Sun) — each column is a 'day card'

Day Card (for each day):
- Shows the date (e.g., 'Mon 10')
- If a meal is set: show meal name, veg/non-veg indicator, 'Edit' button
- If no meal is set: show '+ Add Meal' button
- If the whole day is marked Off (is_off_day = true on BOTH lunch and dinner
  MenuItems for that date): show full-card 'Day Off' badge with undo button
- If just this slot is disabled (is_slot_disabled = true): show 'Slot Disabled'
  badge with undo button, but the other meal type tab still shows normally

Two action buttons at the bottom of each day card:
- 'Disable this slot' — sets is_slot_disabled = true for this meal_type only
  (e.g., disabling Lunch on Wednesday, while Dinner still shows)
- 'Day Off' — sets is_off_day = true on both lunch and dinner MenuItems for
  that date, effectively closing the entire day

Fetch existing menu items via GET /api/vendors/menu?week=2026-02-10
(pass the Monday date of the current week as the week param)
```

```
STEP 4B — Build the Add/Edit Meal slide-over panel for the menu builder.

When clicking 'Add Meal' or 'Edit' on a day card, open a slide-over
panel from the right side of the screen. Panel contains:

- Meal Name (text input, required)
- Description (textarea, max 120 chars, show remaining char count)
- Food Type toggle: Veg | Non-Veg
- Photo upload (1 photo, max 3MB): call POST /api/upload/public-asset,
  store the returned URL in the form state)
- Addons section: shows the vendor's global addon list (fetched from
  GET /api/vendors/addons) as checkboxes. Checked = available that day.

Buttons: Save | Cancel | (if editing) Delete Meal

API routes needed:
- GET /api/vendors/menu?week=YYYY-MM-DD — returns all MenuItems for that week
- POST /api/vendors/menu — create a MenuItem
- PATCH /api/vendors/menu/[id] — update a MenuItem
- DELETE /api/vendors/menu/[id] — delete a MenuItem
- PATCH /api/vendors/menu/[id]/off-day — toggle is_off_day on this MenuItem
  (call for both lunch and dinner IDs when marking a full day off)
- PATCH /api/vendors/menu/[id]/disable-slot — toggle is_slot_disabled on
```

```
  just this MenuItem (single meal type only)
- GET /api/vendors/addons — get vendor's addon list
- POST /api/vendors/addons — create addon

Also build a simple Addons Management page at src/app/(vendor)/menu/addons/page.tsx
Table of addons with: Name, Price, Veg/Non-Veg, Active toggle, Delete button.
Form to add a new addon at the top.
```

Module 5 — Subscription Plans

```
Build the Subscription Plans management page for vendors.

Page at src/app/(vendor)/plans/page.tsx:
- List of existing plans as cards showing: Name, Duration, Meal Type,
  Food Type, Price, Status (Active/Paused)
- 'Create New Plan' button at top right
- Each card has: Edit button, toggle Active/Pause, Delete (only if no
  active subscribers on this plan)

Create/Edit Plan modal/slide-over with fields:
- Plan Name (e.g., 'Weekly Dabba')
- Duration in days (number input, minimum value 4)
- Meal Type: Lunch Only | Dinner Only | Lunch + Dinner
- Food Type: Veg | Non-Veg | Both
- Price (INR, number input)
- Description (short, optional)
- Auto-renewal default: Yes/No toggle

API routes:
- GET /api/vendors/plans — list vendor's plans
- POST /api/vendors/plans — create plan (validate duration >= 4)
- PATCH /api/vendors/plans/[id] — update plan
- PATCH /api/vendors/plans/[id]/toggle — toggle active status
- DELETE /api/vendors/plans/[id] — delete if no active subscribers
```

Module 6 & 7 — Discovery Page & Vendor Public Profile

```
Build the public Discovery page at src/app/(public)/page.tsx

This page is visible without login. It shows a grid of vendor cards.

Filter sidebar (left on desktop, drawer on mobile):
- Food Type: All | Veg | Non-Veg checkboxes
- Meal Type: Lunch | Dinner | Both checkboxes
- Delivery Area: text input for PIN code
- Price Range: min/max inputs

Vendor card shows:
- Cover photo (16:9 aspect ratio)
- Business name
- Veg/Non-Veg badge (green leaf icon for veg, orange icon for non-veg)
- FSSAI Verified badge if is_verified = true
- Cuisine tags (first 3, then '+N more')
- Star rating + review count
- Starting price (lowest plan price)
- Delivery areas (first 2 PIN codes + '+N more')
- 'View Menu' button linking to /vendors/[slug]

API: GET /api/vendors/list with query params for filters
Server-side render this page for SEO.
```

```
Build the Vendor Public Profile page at src/app/(public)/vendors/[slug]/page.tsx

Sections:
1. Hero: Cover photo, name, badges, rating, cuisine tags, about text
2. Delivery Info: PIN codes served, Lunch window, Dinner window
3. Weekly Menu Calendar: Read-only version of the menu builder grid.
   Week navigator. Each day card shows meal name + veg/non-veg badge.
   Clicking a day card expands to show full description + available addons.
   Show 'Menu not published yet' for unpublished weeks.
4. Subscription Plans: Cards for each active plan with name, duration,
   meal type, price. 'Subscribe' button on each.
5. Reviews: Star summary, list of reviews (paginated, 5 per page),
   reviewer name, date, rating stars, comment, vendor response if any.

API: GET /api/vendors/[slug] — returns vendor profile, active plans,
current week menu, and first page of reviews.

Server-side render for SEO. Include meta title and description tags.
```

Module 8 — Payments & Subscriptions

This is the most sensitive module. Read the full prompt before running it, and test thoroughly in Razorpay test mode.

```
Build the subscription checkout and payment flow.

1. Checkout page at src/app/(customer)/checkout/page.tsx
   (receives planId as a query param)
   Shows:
   - Plan summary: vendor name, plan name, duration, meal type, price
   - Wallet credit section: if user has wallet balance, show 'Apply INR X
     credits' toggle. Update total accordingly.
   - Total amount
   - 'Pay Now' button

2. On 'Pay Now':
   - Call POST /api/payments/create-order to create a Razorpay order
   - Open Razorpay checkout modal using the order_id returned
   - On payment success, call POST /api/payments/verify with
     razorpay_order_id, razorpay_payment_id, razorpay_signature

3. API route POST /api/payments/create-order:
   - Require authenticated user
   - Fetch the plan and its vendor from database
   - CAPACITY CHECK: if vendor.active_subscriber_count >= vendor.daily_capacity,
     return 409 error: 'This vendor is currently at capacity.'
     Do this check here (before charging) not just on the frontend.
   - Calculate: total = plan.price - wallet_credits_applied
   - platform_fee = total * commission_rate (from env)
   - Create Razorpay order using razorpay.orders.create()
   - Create a pending Payment record in database
   - Return razorpay order_id and amount
```

```
4. API route POST /api/payments/verify:
   - Verify Razorpay signature using crypto.createHmac
   - If valid:
     * Re-check capacity one more time (race condition guard) — if now at
       capacity, refund via Razorpay and return error
     * Update Payment record status to SUCCESS
     * Deduct wallet credits if applied (create WalletTransaction record)
     * Create Subscription record with:
       start_date = today
       end_date = today + plan.duration_days
       status = ACTIVE
       meals_remaining = plan.duration_days
     * Increment vendor.active_subscriber_count by 1
     * Create daily Order records for each day of the subscription
       (one per meal_type per day — if plan is BOTH, two orders per day)
       Skip order creation for days where the corresponding MenuItem has
       is_off_day = true OR is_slot_disabled = true for that meal_type.
     * Send payment receipt email via Resend
   - Redirect to /subscriptions

5. Also update PATCH /api/subscriptions/[id]/cancel to decrement
   vendor.active_subscriber_count by 1 when a subscription is cancelled.
   Same for when a subscription expires naturally (handle in a daily cron job).

IMPORTANT: Never trust the amount from the client. Always recalculate
the amount server-side from the plan record in the database.
```

⚠ Watch Out

Payments are where bugs cost real money. After building this, test the following manually: successful payment, failed payment, Razorpay modal closed without paying, applying wallet credits, double-submission prevention. Do not proceed to the next module until all these work correctly in test mode.

Module 9 — Customer Dashboard

```
Build the Customer Dashboard at src/app/(customer)/dashboard/page.tsx

Sections:

1. Today's Deliveries
   - Cards for each delivery scheduled today (from Orders table where
     delivery_date = today and user_id = current user)
   - Each card: vendor name, meal type (Lunch/Dinner), delivery status
     badge, delivery window time, any addons for today
   - 'Report Issue' button if status is DELIVERED or NOT_DELIVERED

2. Active Subscriptions
   - Users can have multiple active subscriptions simultaneously (different
     vendors, different plans). Show ALL of them as separate cards.
   - Each card shows: vendor name + cover photo thumbnail, plan name,
     days remaining, next delivery date, meal type badge
   - Action buttons: Pause, Skip Today, Cancel
   - Group cards by vendor name alphabetically if more than one active.
   - Today's deliveries across all vendors are shown in 'Today's Deliveries'
     section above, aggregated into a single unified list.
```

```
3. Pause Subscription flow:
   - Modal asking: pause for how many days? (1–14, number input)
   - Confirm button calls PATCH /api/subscriptions/[id]/pause
   - API: set status to PAUSED, store resume_date = today + pause_days,
     extend end_date by pause_days

4. Skip Today flow:
   - Confirm dialog: 'Skip today's delivery? You'll receive a wallet credit.'
   - Calls PATCH /api/subscriptions/[id]/skip-today
   - API: update today's Order status to NOT_DELIVERED (reason: SKIPPED),
     add wallet credit = (plan.price / plan.duration_days) to user wallet,
     create WalletTransaction record

5. Cancel Subscription flow:
   - Confirm dialog explaining refund policy
   - Calls PATCH /api/subscriptions/[id]/cancel
   - API: update status to CANCELLED, calculate refund credit
     (if remaining_days > 3: credit 50% of remaining value to wallet),
     create WalletTransaction record

4. Wallet Balance widget: shows current wallet balance, last 5 transactions

5. Past Subscriptions: table of expired/cancelled subscriptions
```

Module 10 — Delivery Status System

```
Build the delivery status system for vendors and users.

VENDOR SIDE:
Create page src/app/(vendor)/orders/page.tsx

- Date picker at top (defaults to today)
- List of all Orders for selected date for this vendor
- Each order card shows: subscriber name, address (from subscription
  delivery notes), meal type, addons ordered, current status badge
- Status update buttons: 'Out for Delivery' | 'Mark Delivered' | 'Not Delivered'
  (show contextually based on current status)
- Clicking 'Not Delivered' asks for a reason (dropdown: User Not Home /
  Address Not Found / Other)

API:
- GET /api/vendors/orders?date=YYYY-MM-DD — returns orders for that date
- PATCH /api/orders/[id]/status — update order status
  * On DELIVERED: send push/email notification to user
  * On NOT_DELIVERED: send notification, flag for potential dispute

USER SIDE:
- On the Customer Dashboard, each Today's Delivery card shows a
  status timeline: Preparing → Out for Delivery → Delivered
- Poll GET /api/orders/today every 60 seconds to refresh status
  (WebSockets via socket.io can be added in Phase 2 for true real-time)
```

7. How to Get the Best Results from Cursor

These are the patterns that consistently produce better output from Cursor AI. Apply them throughout the build.

7.1 The Golden Rules

6. One module at a time. Never ask Cursor to build two features in the same prompt.
7. Always provide context. Start prompts with 'I'm building Dabbaz, a tiffin subscription marketplace' and reference the relevant Prisma models.
8. Tell Cursor what NOT to do. Include constraints like 'do not use any UI library', 'do not use localStorage', 'do not create new npm packages'.
9. Ask for the API route and the UI separately. Build the API first, verify it works in Postman/Thunder Client, then build the UI against it.
10. After Cursor generates code, read it. Don't just click 'Apply All'. Read every file and ask Cursor to explain anything you don't understand.

7.2 Debugging Pattern

When something breaks, use this exact approach with Cursor:

```
This code is failing. Here is the exact error:
[paste full error message and stack trace]

Here is the relevant code:
[paste the specific function or file that's failing]

The expected behaviour is: [describe what should happen]
The actual behaviour is: [describe what is happening]

Do not change anything outside of the failing function.
```

7.3 When Cursor Goes Off Track

Cursor sometimes starts inventing things not in your schema, or creates duplicate code, or drifts from the architecture. Signs of this:

- It creates new files you didn't ask for
- It uses a different database pattern than Prisma
- It installs new packages without asking
- The generated code references models that don't exist in your schema

When this happens, stop and use this reset prompt:

```
Stop. Do not generate any more code.

Summarize what you understand about:
```

```
1. The database schema (list the main tables)
2. The tech stack we're using
3. The folder structure of the project

After you summarize, I'll correct any misunderstandings before we continue.
```

7.4 Useful Cursor Commands

- CMD+L — Open Cursor chat panel
- CMD+K — Inline edit (select code first, then press CMD+K to edit just that selection)
- CMD+Shift+L — Add selected file to chat context
- @filename — Reference a specific file in your prompt
- @Codebase — Search across your entire codebase (use sparingly, it slows Cursor down)

Pro Tip
Use CMD+K for small targeted edits (fix a bug, rename a variable, add a field). Use CMD+L for larger feature generation. Mixing them correctly will save you a lot of time.

8. What NOT to Delegate to Cursor

Cursor is a very capable assistant, but there are specific parts of this project where using it blindly can cause serious problems. Handle these yourself.

8.1 Razorpay Webhook Verification

Razorpay sends webhooks to your server when payments succeed, fail, or are refunded. The signature verification code must be exact. A bug here means you'll credit subscriptions for failed payments or miss successful ones. Write this yourself following Razorpay's official documentation, not Cursor's interpretation of it.

```
# Razorpay webhook signature verification — write this yourself
# Docs: https://razorpay.com/docs/webhooks/validate-test/

import crypto from 'crypto'

export function verifyRazorpayWebhookSignature(
  body: string,
  signature: string,
  secret: string
): boolean {
  const expectedSignature = crypto
    .createHmac('sha256', secret)
    .update(body)
    .digest('hex')
  return expectedSignature === signature
}
```

8.2 API Authorization — Verify It Manually

JWT authentication and role middleware protect your routes at the entry point, but you must also verify that controllers only return data belonging to the requesting user. Cursor often writes queries without a user-scoped WHERE clause, which means any authenticated user could access another user's data. After Cursor generates any controller, manually check:

11. Log in as User A, create a subscription.
12. Log in as User B, call GET /api/subscriptions directly with User B's token.
13. Verify User B's response does NOT include User A's subscription.

The fix is always simple — add a WHERE userId = req.user.id to the Prisma query. But catching it early is critical. Check every GET endpoint that returns user-specific data.

8.3 Environment Variables

Never let Cursor create or modify your .env.local file. Never paste real API keys into Cursor prompts. Use placeholder variable names in your prompts (e.g., process.env.RAZORPAY_KEY_SECRET) and fill in the real values yourself.

8.4 Database Migrations

When you need to modify the schema after your first migration, do not let Cursor directly edit migration files. Always make changes to schema.prisma first, review the diff, then run the migration command yourself. Running a migration in the wrong order on a production database can corrupt data.

```
# Always do this in order:
# 1. Edit prisma/schema.prisma
# 2. Preview the migration (don't apply yet)
npx prisma migrate dev --name your_change_name --create-only
# 3. Review the generated SQL in prisma/migrations/
# 4. If it looks right, apply it
npx prisma migrate dev
```

9. Testing Checklist Before You Launch

Before going live, manually test every item below. Do not skip any of them.

Auth

- Sign up with email → verify email → log in → correct role assigned
- Sign up with Google → user created in database with correct role
- Logged out user cannot access /dashboard or /vendor-dashboard
- Non-vendor user cannot access vendor dashboard routes
- Non-admin user cannot access admin routes

Vendor Onboarding

- Cannot submit onboarding form without phone OTP verification
- Cannot submit the same phone number twice
- reCAPTCHA score is stored on the application record
- After approval, user role changes to VENDOR
- After approval, vendor can access vendor dashboard
- Rejection email is received by applicant

Menu & Plans

- Vendor can create, edit, and delete meals
- Vendor can disable just lunch or just dinner on a specific day — the other slot remains unaffected
- Vendor can mark a full day as Off — both lunch and dinner show as Off to users
- Disabled slots and off-day slots are not charged to the subscriber (no Order record created for those days)
- Plan with duration < 4 days cannot be created
- Retired plan still shows for existing subscribers until expiry
- Vendor can set a daily capacity cap and the 'At Capacity' badge appears correctly on listing
- Subscribing when vendor is at capacity returns an error — no charge is made
- active_subscriber_count increments on successful subscription, decrements on cancellation

Payments

- Successful payment creates Subscription + Order records
- Failed payment does NOT create Subscription
- Wallet credits are deducted correctly and the delta is charged to Razorpay
- Razorpay signature verification passes on valid payment
- Razorpay signature verification fails and rejects on tampered data
- Payment receipt email is received

Subscriptions

- User can hold active subscriptions with two different vendors simultaneously
- Each subscription's pause, skip, and cancel logic operates independently
- Today's deliveries from multiple vendors all appear in the unified delivery list
- Pause extends end_date by pause days
- Skip today adds correct wallet credit
- Cancel with > 3 days remaining adds 50% credit to wallet
- Wallet transaction records exist for all credit/debit events

Delivery

- Vendor can update delivery status for each order
- User sees updated status within 60 seconds
- 'Delivered' status triggers user notification

10. Deploying to Azure

You have two deployments to make: the React frontend (Azure Static Web Apps) and the Node.js backend (Azure App Service). Do them in this order.

10.1 Deploy the Backend (Azure App Service)

14. Push your dabbaz-backend repo to GitHub.
15. In Azure Portal → App Services → Create. Runtime: Node 20 LTS. OS: Linux. Plan: Basic B1.
16. In the App Service → Configuration → Application Settings, add every variable from your backend .env file. This is the production equivalent of your .env.
17. In Deployment Center → Source: GitHub. Select your repo and branch (main).
18. Azure will auto-deploy on every push to main.
19. Note your App Service URL (e.g., https://dabbaz-api.azurewebsites.net). This is your VITE_API_BASE_URL for the frontend.
20. Run the production database migration from your local machine pointing at the production DATABASE_URL:

```
# Run once to apply all migrations to production MySQL
DATABASE_URL=your_production_mysql_url npx prisma migrate deploy
```

10.2 Deploy the Frontend (Azure Static Web Apps)

21. Push your dabbaz-frontend repo to GitHub.
22. In Azure Portal → Static Web Apps → Create. Source: GitHub. Select repo and branch.
23. Build details: App location: / — Build command: npm run build — Output location: dist
24. In Static Web Apps → Configuration → Application Settings, add your frontend .env variables (VITE_API_BASE_URL, VITE_RAZORPAY_KEY_ID, VITE_RECAPTCHA_SITE_KEY). Note: these are baked into the build, so a re-deploy is required if you change them.
25. Azure will give you a URL like https://white-river-0123.azurestaticapps.net. This is your production frontend URL.

10.3 Post-Deployment Configuration

26. In Google Cloud Console → OAuth Credentials: add your App Service callback URL to Authorised Redirect URIs (e.g., https://dabbaz-api.azurewebsites.net/api/auth/google/callback).
27. Update GOOGLE_CALLBACK_URL in App Service Application Settings to the production URL.
28. Update FRONTEND_URL in App Service Application Settings to the Static Web Apps URL.
29. In Razorpay Dashboard → Settings → Webhooks: add your App Service webhook URL (e.g., https://dabbaz-api.azurewebsites.net/api/webhooks/razorpay).
30. Switch Razorpay keys from test to live. Update RAZORPAY_KEY_ID and RAZORPAY_KEY_SECRET in App Service settings. Update VITE_RAZORPAY_KEY_ID in Static Web Apps and redeploy.

⚠ Watch Out

Switch to Razorpay LIVE keys only after you have manually end-to-end tested a complete payment flow in production using a test card. Monitor your Razorpay dashboard and Azure App Service logs for the first 24 hours after going live.

DABBAZ — Cursor Build Guide v1.2

Keep this open alongside Cursor. Build one module at a time.