DABBAZ

Vibe Coding Guide with Cursor

Step-by-Step Build Guide

How to build Dabbaz from scratch using Cursor AI

Version 1.3 | 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 (Phase 2)	console.firebase.google.com
Interakt or Wati	WhatsApp Business API for dispatch notifications to customers	interakt.shop or wati.io

> ⚠ **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
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

# 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_xxxxxxxxxxx
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), 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),
  google_id (String?, for Google OAuth users), email (String?, unique?),
  created_at, updated_at
  Note: no password field — auth is phone+OTP or Google OAuth only.

UserAddress: id, user_id (User), label (String, e.g. 'Home'),
  line1 (String), line2 (String?), city (String), pincode (String),
  is_default (Boolean default false), created_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,
  price (Decimal) — price is per individual meal slot, set per day,
  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 | DISPATCHED | DELIVERED | NOT_DELIVERED |
  CANCELLED_BY_CUSTOMER | CANCELLED_BY_VENDOR),
  is_trial (Boolean default false),
  is_frozen (Boolean default false),
  cancelled_at (DateTime?), cancellation_fee (Decimal?),
  dispatched_at (DateTime?),
  delivery_note (String?),
  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), category (TOP_UP | CANCELLATION_CREDIT |
  VENDOR_CANCELLATION_CREDIT | REFUND_COMPLAINT | PAYMENT_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

CartItem: id, user_id (Int?, nullable for pre-login localStorage sync),
  session_id (String?, for pre-login cart identification),
  vendor_id (VendorProfile), menu_item_id (MenuItem),
```

```
  meal_type (LUNCH | DINNER), delivery_date (DateTime),
  addons (Json, array of addon ids and quantities),
  expires_at (DateTime, 7 days from created_at), created_at, updated_at
  @@unique([user_id, menu_item_id])

Invoice: id, user_id (User), invoice_number (String unique),
  type (PER_TRANSACTION | MONTHLY_STATEMENT | VENDOR_SETTLEMENT),
  period_start (DateTime?), period_end (DateTime?),
  subtotal (Decimal), taxable_value (Decimal),
  cgst (Decimal), sgst (Decimal), total_gst (Decimal),
  total_amount (Decimal), pdf_url (String?),
  email_sent_at (DateTime?), payment_id (Int?),
  vendor_payout_id (Int?), created_at

GSTRate: id, name (String), rate (Decimal),
  taxable_percentage (Decimal), sac_code (String),
  is_active (Boolean default true), created_at, updated_at

PlatformSettings: id (always 1, single-row config table),
  wallet_topup_enabled (Boolean default true),
  cancellation_fee (Decimal default 20),
  order_freeze_hour (Int default 21, 24h IST, e.g. 21 = 9 PM),
  whatsapp_enabled (Boolean default true),
  updated_at
  Note: this table always has exactly one row (id=1).
  Use upsert to update it. Never insert a second row.
```

> ⚠ **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 Phase 2 Schema Additions — Collection & Delivery Feature

The following fields are not in the MVP schema. When Phase 2 begins, run a Prisma migration to add them. Use this prompt:

```
I'm building Dabbaz. I need to add Phase 2 fields for the Collection &
Delivery Fulfilment feature. Update prisma/schema.prisma with the
following additions. Do not remove or rename any existing fields.
After updating the schema, write the migration command.

VendorProfile — add these nullable fields:
  delivery_charge (Decimal?, nullable — null means delivery not configured)
  collection_enabled (Boolean default false)
```

```
  collection_address_lunch (String?)
  collection_address_dinner (String?)
  collection_lunch_window_start (String?  — stored as 'HH:MM')
  collection_lunch_window_end (String?)
  collection_dinner_window_start (String?)
  collection_dinner_window_end (String?)

MenuItem — add these fields:
  fulfilment_types (enum FulfilmentType: DELIVERY | COLLECTION | BOTH,
    default DELIVERY)
  max_orders (Int?, nullable — no cap if null)
  max_portions (Int?, nullable — no cap if null)
  current_orders (Int default 0)
  current_portions (Int default 0)

CartItem — add these fields:
  quantity (Int default 1)
  fulfillment_mode (enum FulfilmentMode: DELIVERY | COLLECTION,
    default DELIVERY)

Order — add these fields:
  quantity (Int default 1)
  fulfillment_mode (enum FulfilmentMode: DELIVERY | COLLECTION,
    default DELIVERY)
  delivery_charge (Decimal default 0
    — stores the delivery charge for this delivery event at time of order)

Order status enum — add new values:
  READY_FOR_COLLECTION
  COLLECTED        (reserve for Phase 3, add now to avoid future migration)
  NOT_COLLECTED    (reserve for Phase 3, add now to avoid future migration)

Add a new GSTRate row in a seed file:
  name: 'Delivery Service', rate: 18.00, taxable_percentage: 100.00,
  sac_code: '996812', is_active: true
  (18% GST applies to delivery charges as a logistics service)
```

> **⚠ Watch Out**
>
> Run this migration on a database backup first. The new enum values on Order status are additive and safe. The new fields on CartItem and Order default to safe values. Verify current_orders and current_portions on existing MenuItems are 0 after migration — they should be since this feature was not live.

2.4 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.
Primary auth: phone number + OTP. No email/password.
Alternative auth: Google OAuth.

First, remove bcrypt and passport-local from dependencies —
we are not using email/password.
Keep: passport, passport-google-oauth20, passport-jwt, jsonwebtoken

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.
        phone and phone_verified remain null/false until user adds phone.
      - 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/utils/otp.ts — OTP management using Redis:
   - generateAndStoreOTP(phone): generates a 6-digit OTP,
     stores it in Redis with key 'otp:<phone>' and TTL of 10 minutes,
     returns the OTP string
   - verifyOTP(phone, otp): checks Redis for key 'otp:<phone>',
```

```
     compares value, deletes the key on success, returns boolean
   - rateLimitOTP(phone): checks Redis key 'otp_attempts:<phone>',
     increments it, rejects if > 5 attempts in 1 hour

4. src/routes/auth.routes.ts — Routes:

   POST /api/auth/request-otp
     Body: { phone: string }
     - Validate: 10-digit Indian mobile number
     - Call rateLimitOTP(phone), return 429 if over limit
     - Call generateAndStoreOTP(phone)
     - Send OTP via MSG91 to the phone number
     - Return: { message: 'OTP sent' }
     - No auth required — this is the entry point for all users

   POST /api/auth/verify-otp
     Body: { phone: string, otp: string, name?: string,
             delivery_address?: object }
     - Call verifyOTP(phone, otp), return 400 if invalid
     - Look up User by phone number in the database
     - If user EXISTS: this is a login. Generate tokens. Return user + tokens.
     - If user DOES NOT EXIST: this is registration.
         * name is required in this case — return 400 if missing
         * Create User: phone, phone_verified: true, name, role: CUSTOMER,
           wallet_balance: 0, unique referral_code
         * If delivery_address provided, create UserAddress record
         * Generate tokens. Return user + tokens.
     - Single endpoint handles both login and registration.
       The frontend never needs to know which one happened.

   POST /api/auth/refresh
     - Accept refresh token, validate against RefreshToken table,
       return new access token

   POST /api/auth/logout
     - Delete RefreshToken from DB (requires auth middleware)

   GET  /api/auth/me
     - Return the authenticated user record (requires auth middleware)

   GET  /api/auth/google
     → passport.authenticate('google', { scope: ['email', 'profile'] })

   GET  /api/auth/google/callback
     → on success, generate access + refresh tokens,
       redirect to frontend /auth/callback with tokens as query params

5. src/middleware/auth.middleware.ts
   - requireAuth: passport.authenticate('jwt', { session: false })
   - optionalAuth: same but does not reject if no token present.
     Attaches req.user if token is valid, leaves req.user undefined if not.
     Use this on cart and browsing endpoints.

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

4.3 Build the Frontend Auth

The login page is now a phone+OTP flow. There is no signup page — registration happens automatically at first OTP verification or at cart checkout.

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

1. src/lib/api.ts — Axios instance:
   - baseURL: import.meta.env.VITE_API_BASE_URL
   - On first load, generate a UUID and store in localStorage as
     'cart_session_id' if it doesn't exist yet.
   - Request interceptor:
     * Attach access token from localStorage as 'Authorization: Bearer <token>'
     * Attach 'X-Cart-Session' header with the cart_session_id value
   - Response interceptor: if 401 received, call POST /api/auth/refresh
     with refresh token from localStorage. On success, update access token
     and retry 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 | null), isLoading (boolean)
   - Actions: setUser(user), logout()
   - logout(): clears localStorage tokens, calls POST /api/auth/logout,
     sets user to null
   - On mount: if access token in localStorage, call GET /api/auth/me
     to hydrate user state. Handle 401 by clearing tokens.

3. src/App.tsx — React Router v6 setup:
   - ProtectedRoute: reads from useAuth(), redirects to /login if no user
   - VendorRoute: also checks user.role === 'VENDOR'
   - AdminRoute: also checks user.role === 'ADMIN'
   - Public routes (/ , /vendors/:slug, /how-it-works) are accessible
     to everyone including unauthenticated users
```

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

This is a two-step phone+OTP flow. No email, no password.

Step 1 — Enter Phone:
- Single input: mobile number (10 digits, Indian format)
- react-hook-form + zod validation
- 'Send OTP' button → POST /api/auth/request-otp
- On success, advance to Step 2

Step 2 — Enter OTP:
- Show: 'OTP sent to +91 XXXXXXXXXX'
- 6-digit OTP input (individual boxes or single field — your choice)
- 'Verify' button → POST /api/auth/verify-otp { phone, otp }
- On success: store access_token and refresh_token in localStorage,
  call setUser() with returned user, redirect to /dashboard
- 'Resend OTP' link (available after 30 seconds, calls request-otp again)

Below Step 1, show: 'Or sign in with Google' button
  → window.location.href = VITE_API_BASE_URL + '/auth/google'
  After redirect back, parse tokens from URL query params,
  store in localStorage, call setUser(), redirect to /dashboard.

Handle the /auth/callback route in App.tsx to do this token parsing.

No signup link needed — first-time users are handled automatically.
Tailwind styling, mobile-first, no UI library.
```

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	Approve vendors, manage platform config (cancellation fee, freeze time, wallet toggle).
3	Vendor Profile Setup	Approved vendors configure their profile and delivery areas.
4	Menu Builder	Per-day, per-meal-type menu with individual pricing. Publish/unpublish. Day Off.
5	Public Discovery Page	Pincode filter, vendor cards, browsing without login.
6	Vendor Public Profile Page	Two-week menu calendar, add-to-cart buttons, lunch/dinner slots.
7	Cart with Calendar View	Multi-vendor cart, min-3 rule, calendar view, session persistence.
8	Cart Checkout & Account Creation	Checkout form, phone OTP, account creation, payment method selection.
9	Payments & Wallet	Razorpay integration, wallet top-up toggle, wallet payment, order creation.
10	Order Cancellation	Day-level customer cancellation, 9 PM freeze cron, cancellation credit to wallet.
11	Vendor Order Dashboard	Today's orders list, Mark as Dispatched with WhatsApp + email notification.
12	Vendor Day Off	Mark day as Off, auto-credit affected customers, vendor cancellation flow.
13	Customer Order History	Past orders, active orders, day-level cancel button with countdown to freeze.
14	How It Works Page	Static marketing page. Build last — content can be finalised once platform is live.

6. Cursor Prompts — Module by Module

Below are the exact Cursor prompts to use for each module. Module numbers match the build order table in Chapter 5. Two Phase 2 callouts are included in this chapter (Subscription Plans and Subscription Payments) — these are noted as 'Phase 2 Feature' and should be skipped during the MVP build. Resume them when Phase 2 begins.

Build order reminder

Follow the sequence in Chapter 5. Each module prompt below is numbered to match. Phase 2 callouts are clearly labelled — skip them for now, they are here for completeness.

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.
```

Phase 2 Feature — Subscription Plans (Not in MVP)

Subscription plans are not built in the MVP. The MVP is entirely cart-based — users pick specific days and pay once. The SubscriptionPlan and Subscription tables exist in the schema for forward compatibility, but no UI or API routes are built for them now.

When Phase 2 begins, this module will cover: vendor-facing plan builder (name, duration, meal type, price, auto-renewal toggle), customer-facing Subscribe button on vendor profile, and the subscription billing flow. A dedicated Cursor prompt will be written at that stage.

Module 5 & 6 — 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 (FSSAI Verified, Veg/Non-Veg),
   cuisine tags, about text, delivery pincodes
2. Delivery Info: PIN codes served, Lunch delivery window, Dinner window
3. Weekly Menu Calendar (this is the core of the page):
   - Week navigator (back/forward arrows, current week + next week)
   - Each day shows two rows: Lunch and Dinner
   - Each meal slot card shows: meal name, food type badge (Veg/Non-Veg),
     price, short description
   - Clicking the card expands it to show full description and available addons
   - Each published, available meal slot has an 'Add' button.
     Clicking Add adds that slot to the cart (calls POST /api/cart/items)
     and updates the cart icon badge count.
     If the slot is already in cart, the button reads 'Added ✓' and is disabled.
   - Unpublished slots show 'Menu not published yet' — no Add button
   - Off-day slots show 'Day Off' — no Add button
4. Per-vendor cart progress widget (sticky on mobile, sidebar on desktop):
   - Shows: 'X of 3 meals added this week from [Vendor Name]'
   - Progress bar filling as user adds meals
   - This helps the user understand the minimum before they go to checkout
```

```
API: GET /api/vendors/[slug] — returns vendor profile,
current week menu items, and next week menu items.

Note: Subscription Plans section is Phase 2. Do not add a Subscribe button.
Reviews section is also Phase 2. Include a placeholder 'Reviews coming soon' block.
```

Phase 2 Feature — Subscription Payments (Not in MVP)

Subscription-based payments (recurring billing, creating Subscription records, auto-renewal, meals_remaining counter) are not built in the MVP. The MVP payment flow is cart-based and is covered by the Cart Checkout module — see 'Module 8 — Cart Checkout & Account Creation' and 'Module 9 — Payments & Wallet' in the sections below.

When Phase 2 begins, this module will cover: plan purchase flow with Razorpay, Subscription record creation, daily Order record generation for the subscription period, capacity checks, and subscription management (pause/skip/cancel with wallet credit rollover). The Payment, Subscription, and Order tables are already in the schema ready for this.

Module 7 — Cart with Calendar View

Build in two sub-steps: backend API first, then the calendar UI.

```
STEP 7A — Cart API in dabbaz-backend.

Build these routes in src/routes/cart.routes.ts:

POST /api/cart/items
  Body: { vendor_id, menu_item_id, meal_type, delivery_date }
  - If user is authenticated (req.user exists via optionalAuth): save
    CartItem with user_id
  - If not authenticated: read 'X-Cart-Session' header (UUID generated
    by frontend, stored in localStorage) and save with session_id,
    user_id = null
  - Upsert: if item for same user/session + menu_item_id already exists,
    update it rather than error
  - Set expires_at = now + 7 days
  - Return the cart item

GET /api/cart
  - Auth: optionalAuth (works for both logged-in and session users)
  - If authenticated: return all CartItems for user_id
  - If not: return CartItems for session_id from header
  - Join MenuItem (name, meal_type, price, food_type) and
    VendorProfile (business_name, cover_photo_url)
  - Exclude expired items (expires_at < now)
  - Group by vendor in the response

DELETE /api/cart/items/:id
  - Remove one cart item. Verify ownership (user_id or session_id).

DELETE /api/cart
  - Clear entire cart for user or session. Called after payment success.

GET /api/cart/validate
```

```
  - Requires auth (called at checkout after OTP verification)
  - Group items by vendor_id and calendar week (Mon-Sun of delivery_date)
  - Count meals per vendor per week
  - Return: { qualifying: [{vendorId, vendorName, count}],
              failing: [{vendorId, vendorName, count, needed}] }

POST /api/cart/merge
  - Body: { session_id }
  - Requires auth
  - Move all CartItems with that session_id to req.user.id
  - On conflict (same menu_item_id), keep the user's existing version
```

```
STEP 7B — Cart UI in dabbaz-frontend.

1. src/lib/api.ts additions:
   - On first load: if localStorage has no 'cart_session_id',
     generate a UUID and store it
   - Add 'X-Cart-Session' header to every Axios request
   - After OTP verify/login: call POST /api/cart/merge with the session_id,
     then delete 'cart_session_id' from localStorage

2. Cart icon in navbar:
   - Calls GET /api/cart on mount, shows item count badge
   - Updates on every add/remove
   - Clicking opens CartSidebar

3. CartSidebar (slide-in from right, full height):
   - Two-week calendar at top. Days with cart items get a dot.
   - Below calendar: items grouped by vendor, then sorted by date.
   - Each item row: date chip, meal type badge (L/D), meal name,
     vendor name, price, Remove button
   - Per-vendor progress bar: 'X of 3 meals this week from [Vendor]'
     Green when X >= 3, amber when X < 3
   - Sticky footer: item count, subtotal, 'Proceed to Checkout' button
   - 'Proceed' disabled if any vendor has < 3 meals in a week
     Show tooltip: 'Add X more from [VendorName] or remove them'

4. Add to Cart on VendorProfilePage:
   - Menu displayed as a two-week calendar grid
   - Each cell: meal name, food type dot, price, 'Add' button
   - If slot is not published: show 'Menu coming soon' (greyed)
   - If slot is is_off_day: show 'Closed' (greyed)
   - If item already in cart: 'Added ✓' button (clicking removes it)
   - Successful add: update cart icon badge
```

Module 8 — Cart Checkout & Account Creation

```
Build the checkout flow at src/pages/customer/CheckoutPage.tsx
Route: /checkout

STEP 1 — Vendor Validation:
- Call GET /api/cart/validate (if authenticated) or show the cart
  summary and proceed — validation runs server-side at payment time
- Display per-vendor status:
    ✅ Vendor A — 4 meals (qualifies)
    ⚠️  Vendor B — 2 meals (need 1 more or remove)
- 'Fix' links: 'Add more from Vendor B' → back to vendor profile
```

```
  'Remove Vendor B' → DELETE all Vendor B items from cart
- 'Continue to Details' button: only enabled when all vendors qualify
  OR all non-qualifying vendors have been removed

STEP 2 — Your Details:

IF USER NOT LOGGED IN:
  Form fields: Full Name, Mobile Number (10 digits), Delivery Address
  (Line 1, Line 2 optional, City, PIN Code)
  'Send OTP' button → POST /api/auth/request-otp { phone }
  After OTP sent: 6-digit OTP input appears
  'Verify & Continue' → POST /api/auth/verify-otp
    { phone, otp, name, delivery_address }
  On success: store tokens, call POST /api/cart/merge, delete
  session_id from localStorage, update useAuth(), advance to Step 3
  If phone already has an account: log them in silently, merge cart,
  show 'Welcome back [name]!' and proceed — do not interrupt the flow

IF USER ALREADY LOGGED IN:
  Show saved addresses as selectable cards
  'Add new address' option
  'Continue to Payment' → Step 3

STEP 3 — Payment:
- Itemised order summary: each meal grouped by vendor and date
- GST breakdown: subtotal, taxable value (60%), CGST (2.5%),
  SGST (2.5%), total
- Wallet balance display (if user has balance)
  Payment method selector — three options:
    [Wallet] — shown with current balance. DISABLED with tooltip
    'Insufficient balance' if balance < order total
    [Card / Net Banking] — Razorpay standard checkout
    [UPI] — Razorpay UPI flow
- 'Pay ₹X' button → creates Razorpay order via POST /api/payments/create
  Opens Razorpay modal
  On Razorpay success → POST /api/payments/cart-verify
  On success: redirect to /orders with toast 'Order placed!'

BACKEND — POST /api/payments/create
  - Require auth
  - Re-run vendor minimum validation server-side
  - Calculate total from CartItems (join MenuItem.price)
  - Apply GST: calculateGST(total) from src/utils/gst.ts
  - If payment_method = WALLET: verify wallet_balance >= total,
    debit wallet, create WalletTransaction (PAYMENT_DEBIT),
    skip Razorpay, go straight to order creation
  - If Card/UPI: create Razorpay order, return order_id + key

BACKEND — POST /api/payments/cart-verify
  - Verify Razorpay signature
  - Re-validate vendor minimum (server-side final gate)
  - For each qualifying CartItem: create Order record
    (status=PREPARING, user_id, vendor_id, delivery_date, meal_type,
     price from MenuItem, delivery address from UserAddress)
  - Remove non-qualifying vendor items silently
  - Create Payment record (amount, method, razorpay_id, cgst, sgst)
  - Call DELETE /api/cart to clear cart
  - Send order confirmation email to customer
  - Send new order notification email to each affected vendor
  - Return success
```

Module 9 — Payments & Wallet

```
Build the wallet top-up flow. Only accessible when
PlatformSettings.wallet_topup_enabled = true.

FRONTEND — on the customer Orders/Wallet page:
- Call GET /api/platform-settings/public to check wallet_topup_enabled
- If true: show 'Top Up Wallet' button with amount input (min ₹100)
- If false: hide the button entirely

TOP-UP FLOW:
- User enters amount → POST /api/payments/wallet-topup { amount }
- Backend creates Razorpay order for that amount
- Frontend opens Razorpay modal
- On success: POST /api/payments/wallet-topup-verify
  * Verify Razorpay signature
  * Create WalletTransaction: type=CREDIT, category=TOP_UP
  * Update User.wallet_balance += amount
  * Return new balance
- Frontend updates wallet balance display without page reload

BACKEND — GET /api/platform-settings/public
  - No auth required
  - Return only: { wallet_topup_enabled: boolean }
  - Cache in Redis for 5 minutes (use redis.get/set with TTL)
  - Used by frontend to show/hide top-up button
```

Module 10 — Customer Orders & Cancellation

```
Build the Customer Order History page at src/pages/customer/OrdersPage.tsx

SECTIONS:

1. Upcoming Orders
   - All future Order records for the logged-in user where delivery_date
     >= today and status not in (CANCELLED_BY_CUSTOMER, CANCELLED_BY_VENDOR)
   - Group by vendor, then by date
   - Each order row shows: date, meal type (Lunch/Dinner badge), vendor name,
     meal name, price, status badge (PREPARING / DISPATCHED)
   - Show a 'Cancel this day' button IF the delivery_date is tomorrow or later
     AND current time < 9 PM IST today (the freeze threshold)
   - If delivery_date is tomorrow but it is already past 9 PM: show
     'Order locked — cancellation window closed' instead of the button
   - Show a countdown timer for orders that can still be cancelled today:
     'Cancel by 9:00 PM tonight (X hours remaining)'

2. Past Orders
   - All Orders with delivery_date < today, paginated
   - Show date, vendor, meal, amount paid

3. Wallet Widget (sidebar or bottom section)
   - Current wallet balance
   - Last 10 wallet transactions with category badge:
     TOP_UP / CANCELLATION_CREDIT / VENDOR_CANCELLATION_CREDIT /
     REFUND_COMPLAINT / PAYMENT_DEBIT
   - Top Up button (only shown if admin wallet_topup_enabled = true)
```

```
     → Razorpay payment flow for wallet load

CANCELLATION FLOW:
When 'Cancel this day' is clicked:
- Show confirmation modal:
  'Cancel [Vendor Name] [Lunch/Dinner] on [Date]?
   You will receive ₹[meal_price - cancellation_fee] as wallet credit.
   Cancellation fee: ₹[cancellation_fee].'
- On confirm: POST /api/orders/:id/cancel

BACKEND — POST /api/orders/:id/cancel
  - Require auth, verify order belongs to req.user
  - Check order status is PREPARING (not already dispatched/cancelled)
  - Check is_frozen = false (freeze check: compare current IST time against
    platform setting order_freeze_hour for delivery_date - 1 day)
  - If frozen: return 409 'Cancellation window has closed for this delivery'
  - Fetch platform settings (cancellation_fee) from PlatformSettings table
  - credit_amount = order meal price - cancellation_fee
  - Update Order status to CANCELLED_BY_CUSTOMER, set cancelled_at = now,
    set cancellation_fee on the order record
  - Create WalletTransaction: type=CREDIT,
    category=CANCELLATION_CREDIT, amount=credit_amount,
    reference_id=order.id, reason='Cancelled [Date] [MealType]'
  - Update User.wallet_balance += credit_amount
  - Send confirmation email to customer
  - Return updated order and new wallet balance

BACKEND — Nightly Freeze Cron Job
  Runs at 9:00 PM IST every day (scheduled via Bull, use node-cron or
  a Bull repeatable job with cron expression '0 21 * * *' Asia/Kolkata).
  Steps:
  1. Read order_freeze_hour from PlatformSettings
  2. Find all Orders where delivery_date = tomorrow AND
     status = PREPARING AND is_frozen = false
  3. Set is_frozen = true on all of them
  This prevents any further cancellations for the next day.
```

Module 11 — Vendor Order Dashboard & Dispatch

```
Build the Vendor Orders page at src/pages/vendor/OrdersPage.tsx

PAGE LAYOUT:
- Date picker at top (defaults to today). Range: today to +7 days
- For the selected date, show two sections: Lunch Orders and Dinner Orders
- Each order card shows: customer name, customer phone, delivery address,
  any addons, delivery note, status badge
- 'Mark as Dispatched' button per order (only shown if status = PREPARING)
- Once dispatched, show 'Dispatched at [time]' — no further action needed
- Orders cancelled by customer show as greyed out with 'Cancelled' badge
- Orders from vendors marked as Day Off show as 'Day Off' in place of the list

BACKEND — GET /api/vendors/orders?date=YYYY-MM-DD
  - Require auth + VENDOR role
  - Return all Orders for this vendor for the given date,
    including user name, phone, address (from UserAddress via order)
    and addons. Exclude CANCELLED_BY_CUSTOMER and CANCELLED_BY_VENDOR.

BACKEND — PATCH /api/orders/:id/dispatch
  - Require auth + VENDOR role
```

```
  - Verify order.vendor_id = req.user.vendorProfile.id
  - Verify order status = PREPARING
  - Update order: status = DISPATCHED, dispatched_at = now
  - Trigger dispatch notifications (do both in parallel, don't await both):
    a) WhatsApp via Interakt/Wati API:
       Send pre-approved template to customer phone:
       'Your tiffin from [vendor.business_name] is on its way!'
       Only send if PlatformSettings.whatsapp_enabled = true
    b) Email via Resend to customer:
       Subject: 'Your tiffin is on its way!'
       Body: vendor name, meal type, date, delivery address confirmation
  - Return updated order

WHATSAPP SETUP NOTE (add as a comment in the controller):
  Use Interakt (interakt.shop) or Wati (wati.io) for WhatsApp Business API.
  Requires a pre-approved message template submitted to Meta.
  Add WHATSAPP_API_KEY and WHATSAPP_TEMPLATE_ID to backend .env.
  Template approval takes 24-48 hours — do this before launch.
```

Module 12 — Vendor Day Off & Auto-Credit

```
Build the Day Off management in the vendor dashboard.
Add to src/pages/vendor/MenuPage.tsx (or a dedicated availability tab).

UI — Day Off Section:
- Show a 14-day calendar (current + next week)
- Each day shows: how many orders exist for that day, whether it's
  already marked Off
- 'Mark as Off' button per day (only for future days, not today or past)
- When vendor clicks 'Mark as Off' for a day that has orders:
  Show a confirmation modal:
  'You have X orders on [Date]. Marking this day Off will:
   - Cancel all X orders',
   - Credit each customer's wallet in full (no cancellation fee)',
   - Send email notifications to all affected customers.
   Are you sure?'
- Confirm → POST /api/vendors/days-off

BACKEND — POST /api/vendors/days-off
  Body: { date: string (YYYY-MM-DD) }
  - Require auth + VENDOR role
  - Check date is in the future (not today or past)
  - Find all Orders for this vendor on this date where status = PREPARING
  - For each affected order:
    a) Update order status to CANCELLED_BY_VENDOR
    b) Get the meal price from the linked MenuItem
    c) Create WalletTransaction: type=CREDIT,
       category=VENDOR_CANCELLATION_CREDIT, amount=full meal price,
       reference_id=order.id,
       reason='[VendorName] unavailable on [Date]'
    d) Update User.wallet_balance += full meal price
    e) Send email to customer: 'Unfortunately [VendorName] is unavailable
       on [Date]. ₹X has been credited to your Dabbaz wallet.'
  - Mark the MenuItem for that date as is_off_day = true for all slots
  - Return: { orders_cancelled: N, total_credited: ₹X }

  Run all order updates in a Prisma transaction so either all succeed
  or none do. Do not leave partial state.
```

Module 13 — Admin Dashboard & Platform Config

```
Build the Admin dashboard pages. These are accessible only to ADMIN role users.
Use requireAuth + requireRole('ADMIN') middleware on all admin API routes.

PAGE 1 — Vendor Queue: src/pages/admin/VendorQueuePage.tsx
- Table of all VendorOnboardingRequests with status PENDING
- Each row: business name, contact name, pincode, submitted date,
  reCAPTCHA score, links to view uploaded documents (SAS URL from Azure Blob)
- Action buttons: Approve, Reject (with reason field), Request More Info
- Approve → POST /api/admin/vendors/approve/:requestId
  Creates VendorProfile + sets User.role = VENDOR
- Admin can also click 'Create Vendor Manually' → form to directly
  create a User with VENDOR role and VendorProfile without an onboarding request

PAGE 2 — User Management: src/pages/admin/UsersPage.tsx
- Search by phone or name
- User detail panel: name, phone, registration date, order count,
  wallet balance, wallet transaction history
- 'Issue Wallet Credit' button: enter amount, reason → creates
  WalletTransaction with category=REFUND_COMPLAINT

PAGE 3 — Order Lookup: src/pages/admin/OrdersPage.tsx
- Search orders by customer phone, vendor name, or date range
- View order details for support resolution

PAGE 4 — Platform Config: src/pages/admin/PlatformConfigPage.tsx
Form that reads from and writes to the PlatformSettings table (always row id=1).

Fields:
  Wallet Top-Up Toggle (boolean switch)
    Label: 'Allow customers to load money into wallet'
    When OFF: users can still spend existing balance and receive credits,
    but the Top Up button is hidden from customer UI

  Cancellation Fee (number input, INR, e.g. 20)
    Label: 'Customer cancellation fee (₹ flat amount deducted from credit)'

  Order Freeze Time (number input, 0-23, 24h format)
    Label: 'Daily order freeze hour (IST, 24h). Default: 21 (9 PM)'

  WhatsApp Notifications (boolean switch)
    Label: 'Send WhatsApp dispatch notifications to customers'

  Save button → PATCH /api/admin/platform-settings
  Use Prisma upsert with id=1

Important: every place in the backend that uses cancellation_fee or
order_freeze_hour must READ it from PlatformSettings at runtime,
not from a hardcoded constant. Cache it in Redis for 5 minutes
to avoid a DB hit on every order operation.
```

Module 14 — How It Works Page

```
Build the How It Works page in dabbaz-frontend.
File: src/pages/public/HowItWorksPage.tsx
Route: /how-it-works
This is a fully static page — no API calls.
```

```
Sections (build top to bottom):

1. Hero
   - Headline: 'Fresh home-cooked tiffin, delivered daily.'
   - Subheadline: 'Pick your days, pick your vendors. Pay once.
     No subscriptions needed to get started.'
   - CTA button: 'Find a Kitchen Near You' → /

2. How It Works — 3 Steps
   Step 1: Browse — Find verified home kitchens near your office or home.
     Filter by area, veg/non-veg, and cuisine type.
   Step 2: Pick Your Days — Choose any combination of lunch and dinner
     slots across the week. Minimum 3 meals per kitchen.
   Step 3: Get It Delivered — Your tiffin arrives fresh.
     Cancel any day before 9 PM the night before if plans change.

3. Why Dabbaz — 4 trust tiles in a grid:
   - FSSAI Verified Kitchens
   - No Password Needed — just your phone
   - Cancel Any Day Before 9 PM
   - Secure Razorpay Payments

4. For Vendors:
   - Heading: 'Cook what you love. We handle the orders.'
   - 3 points: Publish your weekly menu, manage daily orders from
     your dashboard, get paid directly.
   - 'Become a Kitchen Partner' CTA → /become-a-vendor

5. FAQ accordion (8 questions):
   Q: Do I need to create an account to order?
   Q: What is the minimum number of meals I need to order?
   Q: Can I order from multiple kitchens at once?
   Q: Can I cancel a day if my plans change?
   Q: What happens if the kitchen is unavailable on a day I ordered?
   Q: How do I pay — card, UPI, or wallet?
   Q: What if my food doesn't arrive?
   Q: How do I become a vendor on Dabbaz?

6. Final CTA: 'Ready to order?' → 'Browse Kitchens' button → /

Add 'How It Works' to the main navbar.
Link it from the footer and the empty state of the discovery page.
```

6B. Phase 2 Cursor Prompts — Collection & Delivery Fulfilment

Build these modules after the Phase 2 schema migration in section 2.3 is complete and verified. Follow the same discipline as Phase 1 — one module at a time, API before UI.

Prerequisites

Before starting any module in this section: (1) run the Phase 2 migration from section 2.3, (2) verify all new columns exist in MySQL Workbench, (3) run npx prisma generate to refresh the Prisma client.

Phase 2 Module A — Vendor Fulfilment Settings

```
Add fulfilment configuration to the vendor profile settings in
dabbaz-backend and dabbaz-frontend.

BACKEND — PATCH /api/vendors/profile (extend existing endpoint):
  Accept new fields in the request body:
    delivery_charge (Decimal? — validate: if provided, must be > 0)
    collection_enabled (Boolean)
    collection_address_lunch (String?)
    collection_address_dinner (String?)
    collection_lunch_window_start (String? — validate format HH:MM)
    collection_lunch_window_end (String? — validate format HH:MM)
    collection_dinner_window_start (String?)
    collection_dinner_window_end (String?)

  Server-side validation:
  - If collection_enabled = true, collection_address_lunch and
    collection_lunch_window_start/end are required.
  - delivery_charge, if provided, must not exceed the vendor's
    lowest active MenuItem price. Return a helpful error if it does:
    'Delivery charge cannot exceed meal price (lowest meal: ₹X)'
    Query the vendor's MenuItems to find the minimum price.

FRONTEND — Vendor Profile Settings page, new Fulfilment tab:

Section 1 — Delivery:
  - Number input: 'Delivery charge per delivery event (₹)'
    Helper text: 'Charged once per date × meal type combination,
    regardless of how many portions the customer orders.'
  - Leave blank if you do not offer delivery.

Section 2 — Collection (shown always, fields enabled only when
  collection_enabled toggle is ON):
  - Toggle: 'Offer collection / pickup'
  - Lunch collection address (text input)
  - Lunch collection window: start time + end time (time pickers)
  - Dinner collection address (text input — with note:
    'Can be different from lunch address')
  - Dinner collection window: start time + end time

Save button → PATCH /api/vendors/profile

Also extend the Menu Builder (Module 4) slot editor:
  Add a 'Fulfilment' dropdown per slot: Delivery | Collection | Both
  Default follows vendor profile setting.
  Save via PATCH /api/vendors/menu/[id]
  Only show this dropdown if the vendor has configured at least one
  fulfilment mode (delivery_charge set OR collection_enabled = true).
```

Phase 2 Module B — Dual Capacity Caps in Menu Builder

```
Extend the Menu Builder slot editor to support dual capacity caps.
This replaces the old single daily_capacity field on VendorProfile.

FRONTEND — Menu Builder slot editor (existing slide-over panel),
add a new Capacity section:

  'Max orders for this slot (leave blank for no limit)'
  Number input → saves to MenuItem.max_orders

  'Max portions for this slot (leave blank for no limit)'
  Number input → saves to MenuItem.max_portions

  Helper text below both inputs:
  'Orders = number of distinct customers. Portions = total boxes.
  Whichever limit is hit first will mark this slot as At Capacity.
  Current: X orders, Y portions reserved (including cart reservations).'

  Show current_orders and current_portions as read-only live counts.
  These update in real time from GET /api/vendors/menu?week=YYYY-MM-DD.

BACKEND — POST /api/cart/items (extend existing endpoint):
  Before creating the CartItem, run the capacity check:
  1. Fetch the MenuItem for the requested menu_item_id
  2. If max_orders is set AND current_orders + 1 > max_orders → 409:
     'This slot has reached its order limit.'
  3. If max_portions is set AND current_portions + quantity > max_portions
     → 409: 'This slot has reached its portion limit.'
  4. If both checks pass:
     - Create CartItem with quantity
     - INCREMENT MenuItem.current_orders by 1
     - INCREMENT MenuItem.current_portions by quantity
     - Do both increments in the same Prisma transaction as CartItem create

BACKEND — DELETE /api/cart/items/[id] (extend existing endpoint):
  After deleting CartItem:
  - DECREMENT MenuItem.current_orders by 1
  - DECREMENT MenuItem.current_portions by CartItem.quantity
  - Both decrements in same transaction as CartItem delete

BACKEND — Nightly cart expiry cron (extend existing 7-day cleanup):
  For each expired CartItem being deleted:
  - DECREMENT MenuItem.current_orders by 1
  - DECREMENT MenuItem.current_portions by CartItem.quantity
  - Do decrements in same transaction as CartItem delete
  - Process in batches of 100 to avoid long-running transactions

BACKEND — POST /api/payments/create-razorpay-order (extend existing):
  Before creating the Razorpay order, re-validate caps for all CartItems:
  For each CartItem in the cart:
    Re-fetch MenuItem. Check if current state still fits within both caps.
    Note: the customer's own cart reservation is already counted in
    current_orders and current_portions — do not double-count it.
    Check: (current_orders - 1 + 1) <= max_orders (i.e. current is fine)
    Actually: since their reservation is already included, just check
    current_orders <= max_orders and current_portions <= max_portions.
  If any slot fails: return 409 with slot details. Do NOT create
  Razorpay order. No money moves.

FRONTEND — Vendor profile page, meal slot Add button:
  If MenuItem.current_orders >= max_orders OR
     MenuItem.current_portions >= max_portions:
```

```
  Show 'At Capacity' badge instead of Add button.
  The GET /api/vendors/[slug] response should include
  current_orders, current_portions, max_orders, max_portions
  per slot so the frontend can render this without an extra call.
```

Phase 2 Module C — Portions (Quantity Selector)

```
Add quantity selection to the cart add flow in dabbaz-frontend.

FRONTEND — Vendor profile page, meal slot Add interaction:
  Clicking Add on a slot opens a small selector panel (inline, not
  a modal) showing:
  - Meal name and price
  - Quantity stepper: [ − ] [ 1 ] [ + ] (min 1, max 10 or max_portions
    remaining, whichever is lower)
  - If vendor offers BOTH delivery and collection:
    Two radio buttons:
    ○ Deliver to my address  ₹[meal_price] + ₹[delivery_charge]/delivery
    ○ Collect it myself       ₹[meal_price]
      'Pickup: [window] at [address]'
    If vendor offers only DELIVERY or only COLLECTION: no radio shown,
    mode is implicit.
  - 'Add to Cart' confirm button

  Fulfilment mode is per-vendor per-cart:
  If the customer already has items from this vendor in the cart,
  lock the fulfilment mode to match — do not show the radio.
  Show a note: 'Delivery mode set for all [Vendor Name] items.'
  with a 'Change for all items' link (see vendor group UI below).

FRONTEND — Cart sidebar, vendor group display:
  Each vendor group shows:
  - Fulfilment mode badge: 🚚 Delivery or 🥡 Collection
  - 'Change to Collection' / 'Change to Delivery' link
    On click: show confirmation: 'This will update all X items
    from [Vendor Name] to [mode]. Continue?'
    On confirm: PATCH /api/cart/fulfillment-mode
    { vendor_id, fulfillment_mode } — updates all CartItems
    for this vendor in this cart session.
  - If Delivery: show 'Delivery: ₹[charge] × [N events] = ₹[total]'
    as a separate line under the vendor group
  - If Collection: show '[window] at [address]' as reminder line

BACKEND — POST /api/cart/items (extend):
  Accept quantity (Int, default 1) and fulfillment_mode in request body.
  Validate quantity >= 1.
  Validate fulfillment_mode is valid for this MenuItem.fulfilment_types.
  (e.g. if slot is DELIVERY only, reject fulfillment_mode = COLLECTION)

BACKEND — PATCH /api/cart/fulfillment-mode (new endpoint):
  Body: { vendor_id, session_id_or_user_id, fulfillment_mode }
  Update fulfillment_mode on all CartItems for this vendor in this cart.
  Validate the new mode is offered by the vendor
  (collection_enabled = true for COLLECTION).

BACKEND — GET /api/cart/summary (extend existing or create):
  Returns cart grouped by vendor with:
  - Items with quantity and fulfillment_mode
  - Delivery charge calculation per vendor:
```

```
    delivery_events = count of unique (delivery_date, meal_type)
    combinations among items where fulfillment_mode = DELIVERY
    vendor_delivery_total = delivery_events × vendor.delivery_charge
  - Per-vendor subtotal = sum(item.price × quantity) + vendor_delivery_total
  - Cart grand total = sum of all vendor subtotals
  - GST breakdown:
    food_taxable = total food price × 0.60
    food_gst = food_taxable × 0.05
    delivery_gst = total delivery charges × 0.18
  - meal_units_per_vendor (sum of quantities per vendor per week,
    for minimum-3 progress display)

FRONTEND — Cart sidebar, minimum rule progress update:
  Change 'X of 3 meals this week' to 'X of 3 meal units this week'
  Tooltip: 'Each portion counts as 1 unit toward the minimum.'
```

Phase 2 Module D — Checkout & Payment Updates

```
Extend the checkout flow to handle delivery charges, collection-only
carts, and per-delivery-event charge calculation.

FRONTEND — Checkout page, Step 2 (Your Details):
  Delivery address section is CONDITIONAL:
  - If ALL vendors in cart have fulfillment_mode = COLLECTION:
    Hide address input entirely.
    Show a 'Collection Summary' section instead:
    One card per vendor: vendor name, collection address, window.
  - If ANY vendor has fulfillment_mode = DELIVERY:
    Show address input as normal.
  - Mixed cart: show address input AND collection summary cards.

FRONTEND — Checkout page, Step 3 (Payment):
  Order summary shows itemised per-vendor breakdown:
  For each vendor:
    - Each meal slot: name × quantity, base price
    - If DELIVERY: 'Delivery (N events × ₹X)' as separate line
    - If COLLECTION: 'Collection — [window] at [address]'
    - Vendor subtotal
  Below all vendors:
    - Food GST (3% effective): ₹X
    - Delivery GST (18%): ₹X  (only shown if any delivery in cart)
    - Grand total: ₹X

BACKEND — POST /api/payments/create-razorpay-order (extend):
  Server-side total recalculation must now include delivery charges:
  1. Group CartItems by vendor
  2. For each vendor:
     a. Sum food total: sum(item.price × item.quantity)
     b. If any items have fulfillment_mode = DELIVERY:
        delivery_events = count of unique (delivery_date, meal_type)
        vendor_delivery_charge = delivery_events × vendor.delivery_charge
     c. Vendor total = food total + vendor_delivery_charge
  3. Grand food total = sum of all vendor food totals
  4. Grand delivery total = sum of all vendor delivery charges
  5. food_gst = (grand_food_total × 0.60) × 0.05
  6. delivery_gst = grand_delivery_total × 0.18
  7. amount = grand_food_total + grand_delivery_total +
              food_gst + delivery_gst
  NEVER trust the total from the client. Always recalculate server-side.
```

```
BACKEND — POST /api/payments/verify (extend):
  When creating Order records after successful payment:
  - Set Order.quantity from CartItem.quantity
  - Set Order.fulfillment_mode from CartItem.fulfillment_mode
  - Set Order.delivery_charge:
    For DELIVERY orders: apportion the delivery charge to this order.
    Simple approach: store vendor.delivery_charge on every delivery
    order (not the total — just the per-event rate).
    The vendor dashboard can then show the rate per event.
  - Set Order.status = PREPARING for both delivery and collection orders
```

> **⚠ Watch Out**
>
> After building Module D, test these specific scenarios manually: all-delivery cart, all-collection cart, mixed cart, cart with portions > 1, delivery charge appearing correctly in Razorpay amount, GST on delivery calculated separately from food GST. Do not go live until all pass.

Phase 2 Module E — Vendor Order Dashboard: Collections

```
Extend the Vendor Order Dashboard (Module 11) to handle collection orders.

FRONTEND — Vendor Orders page (src/pages/vendor/OrdersPage.tsx):
  Split the day view into two tabs or two sections:

  Section 1 — Deliveries (existing, extend):
  Each order card now shows quantity: 'Mon 24 Feb Lunch × 2'
  Delivery address shown as before.
  'Mark as Dispatched' fires WhatsApp + email as existing.

  Section 2 — Collections (new):
  Only shown if vendor has collection_enabled = true.
  Each order card shows:
  - Customer name, phone
  - Meal name × quantity
  - Collection window reminder: 'Customer collects by [end_time]'
  - Status badge: PREPARING or READY_FOR_COLLECTION
  - 'Mark as Ready' button (only shown if status = PREPARING)

BACKEND — PATCH /api/orders/:id/ready-for-collection (new endpoint):
  - Require auth + VENDOR role
  - Verify order.vendor_id = req.user.vendorProfile.id
  - Verify order.fulfillment_mode = COLLECTION
  - Verify order.status = PREPARING
  - Update order.status = READY_FOR_COLLECTION
  - Send notification to customer:
    WhatsApp (if whatsapp_enabled): pre-approved template:
    'Your tiffin from [vendor.business_name] is ready for collection.
     Please collect by [collection_window_end] at [collection_address].'
    Email via Resend: same message.
  - Return updated order

BACKEND — GET /api/vendors/orders?date=YYYY-MM-DD (extend):
  Return fulfillment_mode and quantity on each order.
  Frontend uses fulfillment_mode to split into Deliveries / Collections.
```

```
Add 'READY_FOR_COLLECTION' to the notification events table in
section 5.7.2 of the PRD — already documented, just ensure the
WhatsApp template is submitted to Meta for approval before launch.
```

Phase 2 Module F — Customer Orders Page & Discovery Filter

```
Two small updates: customer-facing order display and discovery filter.

FRONTEND — Customer Orders page (extend Module 10):
  Upcoming orders list — each order row now shows:
  - Fulfilment mode badge: 🚚 Delivery or 🏪 Collection
  - If Collection and status = READY_FOR_COLLECTION:
    Show a highlighted banner: 'Ready for collection — pick up by
    [end_time] at [collection_address]'
  - Quantity shown: 'Mon 24 Feb Lunch × 2'
  - Cancel button logic unchanged — cancels whole slot
  - Cancellation modal text update for portions:
    'Cancel [Vendor] [Lunch/Dinner × N] on [Date]?
     You will receive ₹[(meal_price × N) - cancellation_fee] credit.
     Cancellation fee: ₹[fee].'

BACKEND — POST /api/orders/:id/cancel (extend):
  credit_amount = (order.price × order.quantity) - cancellation_fee
  (Previously: credit_amount = order.price - cancellation_fee)
  All other cancellation logic unchanged.

FRONTEND — Discovery page (extend Module 5):
  Add one checkbox to the filter sidebar:
  ☐ Offers collection / pickup
  Maps to: GET /api/vendors/list?collection=true

BACKEND — GET /api/vendors/list (extend):
  Accept optional query param: collection (boolean)
  If collection=true: add WHERE collection_enabled = true to query.
  No other changes needed.
```

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

- New user: enter phone → receive OTP → verify → account created, logged in automatically, role = CUSTOMER
- Existing user: enter same phone → OTP → logged in — no duplicate account created
- OTP expires after 10 minutes — expired OTP returns 400
- More than 5 OTP requests for the same phone in 1 hour returns 429
- Google OAuth: new user created with google_id, can complete checkout after adding phone
- JWT access token auto-refreshes on 401 — user never sees a login redirect mid-session

- Logged-out user cannot access /orders, /checkout, or vendor/admin routes

Vendor Onboarding

- Cannot submit onboarding form without phone OTP verification
- reCAPTCHA score is stored on the application record
- After approval: user role changes to VENDOR, vendor dashboard is accessible
- After rejection: applicant receives rejection email with reason
- Admin can create a vendor manually from admin panel — account is immediately usable

Menu Builder

- Vendor can create a Lunch and Dinner slot separately for each day — different names, prices, descriptions
- Vendor can set different prices on different days (e.g., ₹150 weekday, ₹180 weekend)
- Published meal slots appear on vendor public profile page with correct price
- Unpublished days show 'Menu not yet published' to customers
- Vendor can mark a full day as Off — both slots show as Off on the public profile

Discovery & Browsing

- Public vendor listing loads without login
- Pincode filter returns only vendors whose delivery_pincodes include the searched PIN
- Vendor card shows correct cuisine, food type badge, and delivery area
- Clicking a vendor opens their profile with the two-week menu calendar

Cart

- Unauthenticated user can add meals to cart — items saved under session_id in DB (not just localStorage)
- Cart persists across page refresh and browser close
- Adding both Lunch and Dinner from the same vendor on the same day counts as 2 toward the minimum-3
- Per-vendor progress bar in cart sidebar updates in real time as meals are added/removed
- Cart sidebar 'Proceed to Checkout' is disabled while any vendor has fewer than 3 meals in a week
- Removing all items from a vendor clears that vendor's group from the cart
- Cart items expire after 7 days — the nightly cron deletes expired items

Checkout & Payments

- Vendor minimum validation runs on the checkout page — non-qualifying vendor shows fix-or-remove prompt
- User can remove a non-qualifying vendor and proceed with qualifying vendors only
- Unauthenticated user enters name, phone, address at checkout — OTP sent to that phone
- New user: account created on OTP verify, session cart merged, proceeds to payment without interruption
- Existing user: recognised by phone, logged in silently, cart merged — no duplicate account
- Delivery address saved to UserAddress and linked to each Order
- Wallet payment: only selectable if balance >= order total. Greyed with tooltip if insufficient.
- Wallet payment: deducts correct amount, creates PAYMENT_DEBIT WalletTransaction
- Razorpay payment: signature verification passes on valid payment, rejects tampered data
- Failed Razorpay payment: cart is preserved, user returns to checkout to retry
- Successful payment: Order records created for each qualifying CartItem, cart cleared
- Order confirmation email received by customer
- New order notification email received by each vendor

Wallet & Cancellation

- Admin sets cancellation fee to ₹20 — customer cancels ₹150 meal — receives ₹130 CANCELLATION_CREDIT
- Customer cannot cancel an order with delivery_date tomorrow after 9 PM — order shows as locked
- Customer can cancel an order before 9 PM — order disappears from upcoming list, wallet credit appears
- Wallet transaction list shows correct category badge (CANCELLATION_CREDIT vs TOP_UP vs REFUND_COMPLAINT)
- Wallet top-up button is hidden when admin wallet_topup_enabled = false
- Wallet top-up button is visible when wallet_topup_enabled = true — flow creates TOP_UP transaction

Vendor Day Off & Dispatch

- Vendor marks a day as Off — all affected customers receive email, wallets credited in full (no fee)
- VENDOR_CANCELLATION_CREDIT amount = exact meal price, not meal price minus fee
- Vendor order list shows correct meals for the selected date
- Vendor clicks 'Mark as Dispatched' — customer receives WhatsApp + email notification

- WhatsApp disabled in PlatformSettings — dispatch fires email only, no WhatsApp error

Admin Platform Config

- Changing cancellation fee in Platform Config takes effect on next cancellation (not cached stale value)
- Changing order freeze time to 20:00 — orders freeze at 8 PM that night
- Admin issues REFUND_COMPLAINT credit — correct amount appears in customer's wallet history

---

9B. Phase 2 Testing Checklist — Collection & Delivery Fulfilment

Run through these before launching the Phase 2 feature. Each group maps to one of the Phase 2 modules.

Vendor Fulfilment Settings

- Vendor sets delivery charge ₹20 — field saved, visible on profile page to customers
- Vendor attempts delivery charge of ₹250 on a ₹200 meal — server returns validation error
- Vendor enables collection, sets separate lunch and dinner addresses and windows — all saved correctly
- Vendor sets per-slot fulfilment override to COLLECTION on a specific day — that slot shows collection-only to customers

Dual Capacity Caps

- Vendor sets max_orders = 3 on Monday Lunch — 3rd customer adds to cart, 4th customer sees 'At Capacity' badge
- Vendor sets max_portions = 5 — customer adds 3 portions, next customer adds 3 portions, second add is blocked (would reach 6 > 5)
- Customer removes a slot from cart — current_orders and current_portions decrement immediately, slot becomes available again
- Nightly cron deletes an expired CartItem — current_orders and current_portions decrement correctly
- Checkout initiation: customer has slot in cart, another customer fills remaining capacity while first customer is on checkout page — first customer's checkout is blocked with a clear error, no Razorpay order created

- Both caps null — no capacity enforcement, any quantity accepted

Portions

- Customer orders 1 slot × 3 portions — cart progress shows '3 of 3 meal units', checkout proceeds
- Customer orders 2 slots × 1 portion each — cart shows '2 of 3 meal units', checkout blocked
- Vendor dashboard shows 'Mon 24 Feb Lunch × 3' — vendor knows to pack 3 boxes
- Cancellation of 1 slot × 3 portions — credit = (meal_price × 3) − cancellation_fee

Fulfilment Mode & Cart Display

- Vendor offers BOTH modes — Add button shows delivery/collection radio on vendor profile
- Vendor offers DELIVERY only — no radio shown, delivery assumed silently
- Vendor offers COLLECTION only — no radio shown, collection address and window shown
- Customer picks Delivery for Vendor A's first slot — all subsequent Vendor A slots locked to Delivery, 'Change to Collection' link visible
- Customer clicks 'Change to Collection' for Vendor A — all Vendor A items switch to Collection, delivery charge line disappears
- Cart sidebar shows delivery charge correctly: 3 unique delivery events × ₹20 = ₹60 shown as separate line
- Feb 23 Lunch × 2 portions + Feb 24 Lunch × 1 portion = 2 delivery events = ₹40 (not ₹60)
- Feb 23 Lunch + Feb 23 Dinner = 2 delivery events = ₹40 (not ₹20)

Checkout & Payments

- All-collection cart — address input hidden, collection summary cards shown per vendor
- Mixed cart — address input shown for delivery vendor, collection summary shown for collection vendor
- Grand total includes delivery charges and separate delivery GST (18%)
- Food GST (3% effective) and delivery GST (18%) shown as separate lines
- Server-side total recalculation matches frontend display — tamper with client amount, server rejects
- Order records created with correct quantity, fulfillment_mode, and delivery_charge per order

Collection Vendor Dashboard & Notifications

- Vendor marks collection order as 'Ready for Collection' — customer receives WhatsApp + email with address and window
- Vendor marks delivery order as 'Dispatched' — dispatch notification fires as before, no collection notification
- Collection section only appears on vendor dashboard if collection_enabled = true
- Deliveries and Collections show as separate sections on the day view

Discovery Filter

- 'Offers Collection' checkbox on discovery — only vendors with collection_enabled = true returned
- Unchecking filter — all vendors returned regardless of collection_enabled

---

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 v1.4

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