<u>Mobile App Name: BOMA</u>

1. What this project actually is

You are building a Tanzania-focused short-stay and mid-stay rental marketplace.

Core idea: a mobile app where guests can search and book verified apartments or rooms, and hosts can list and manage their properties. Your platform sits between them and handles:

- Identity and basic KYC
- Bookings and availability
- Payments and payouts using local gateways (AzamPay, Selcom, etc.)
- Deposits and damages
- Cancellations, refunds and disputes
- Reviews and reputation
- Support and communication

Mobile app is the main product. Backend is a single FastAPI application. Database is Postgres on Neon. Authentication is handled by Clerk, but you still maintain your own user profiles and roles. Storage for media (mainly photos) is offloaded to a dedicated media service (Cloudinary). Payments are integrated via your backend to AzamPay and/or Selcom, through a clean abstraction layer, so adding card payments or another provider later does not require redesigning everything.

The project is built for today's Tanzania (mobile money, mid-range smartphones, sometimes unstable network) and also prepared for tomorrow (card payments, AI pricing, smart locks, multi-country expansion).

---

2. Overall architecture in plain language

Think of the system in five main layers:

1. Mobile client: Expo / React Native app, for guests and hosts.
2. Backend API: FastAPI application, exposing REST (and optionally later WebSocket) endpoints.
3. Database: Neon Postgres, holding all structured data.
4. Storage and external services:
    - Media storage and CDN for images and videos.
    - Payment gateways: AzamPay, Selcom, later maybe Stripe, Flutterwave, etc.
    - Notification providers: SMS, email, push notifications.
    - Smart lock providers in the future.

5. Background processing:
   - A worker process (Celery or RQ with Redis) for sending notifications, processing webhooks, running scheduled jobs, and anything that should not block user requests.

Hosting:

- FastAPI and the worker run on a VPS (DigitalOcean, Hetzner or similar), behind a reverse proxy (Nginx or Caddy).
- Neon hosts Postgres.
- Media storage is external (Cloudinary).
- DNS and TLS are handled through something simple like Cloudflare.

Important: it is a monolith in terms of code, not a bunch of microservices. One backend codebase. Separate worker process, but still from the same repository. That keeps things manageable.

---

3. Authentication and identity model

Authentication is delegated to Clerk. This means the Expo mobile app talks directly to Clerk for sign up, login, session management, OTPs, etc. Clerk issues tokens (JWTs) which the mobile app then uses to talk to your FastAPI backend.

On the backend side:

- Every request that requires authentication carries a token.
- FastAPI middleware verifies the token using Clerk's public keys.
- From the token, the backend gets an external user identifier (Clerk user ID) and possibly some basic profile data (email, phone number, etc.).
- The backend maps this external ID to your own internal user record in Postgres.

Your own database will have:

- A `users` table: one row per person who uses the app. It stores:
  - Internal user id (primary key)
  - External auth id (Clerk id)
  - Role flags (guest, host, admin; a user can be both guest and host)
  - Country (start with Tanzania, but keep it flexible)
  - Basic metadata and status fields (active, banned, email verified, phone verified, KYC status, created_at, updated_at).

On top of that:

- A `guest_profiles` table for guest-specific info (preferences, language, emergency contact, identification details).
- A `host_profiles` table for host-specific info (individual vs business, company name, tax info, payout details, KYC documents, verification status).

Clerk handles the heavy lifting of authentication and password/session security. Your system handles:

- Roles and authorisation.
- KYC state.
- Mapping a person to bookings, properties, payouts, and disputes.

This design means you can swap Clerk with a different auth service in the future if you ever have to, without destroying your core data.

---

4. Data model, entities and relationships

You are building a marketplace, so your main entities are Users, Properties, Bookings, and Financial Records. Everything else supports those.

Core entities:

Users and profiles

- `users`: core identity of any person in the system.
- `guest_profiles`: exactly one per user who acts as a guest.
- `host_profiles`: one per user who acts as a host.

Properties and availability

- `properties`: a property that can be booked. It includes:
  - Host who owns/manages it.
  - Address and geolocation.
  - Basic description and amenities.
  - Rules and constraints (max guests, pets allowed, smoking, etc.).
  - Country and city.
  - Default pricing.
  - Verification status.
- `property_units` (optional, depending on design): if a host has multiple identical units in one building, this table can represent each unit separately, or you can fold this into `properties` and treat each property as a single rentable unit.
- `property_photos`:
  - One row per photo, storing the URL, order, type (cover photo or not), whether it is verified (taken by your team) or uploaded by host.

- `property_documents`:
    - Optional, for ownership proof, agreements, and any legal documents. Stored as references to secure storage.
- `amenities` and a join table `property_amenities` if you want configurable amenity types.

Availability and pricing

- `availability_rules`:
    - Rules like minimum nights, maximum nights, specific blocked dates, seasonal adjustments.
- `availability_overrides`:
    - Specific dates manually blocked or opened by host.
- `pricing_rules`:
    - Weekly discounts, monthly discounts, weekend price multipliers.
- These three can be kept simple at first, then extended if you introduce an AI pricing engine later.

Bookings, stays and reviews

- `bookings`:
    - Core of the business.
    - Fields include property, guest, host, check-in date, check-out date, total price, status (pending, awaiting_payment, confirmed, cancelled, completed, no_show, etc.), cancellation policy, deposit amount, payment status, timestamps.
- `stays` (optional): you can either reuse the `bookings` table to represent stays, or have a `stays` table that is derived from booking once guest checks in. Not mandatory in v1.
- `reviews`:
    - Reviews from guests about properties and hosts.
    - Optionally, reviews from hosts about guests (kept private for internal risk assessment).

Financial data

- `payments`:
    - Records each incoming payment from the guest, regardless of gateway.
    - Includes gateway, reference, status, amount, currency, fees, raw payload.
- `payouts`:
    - Records outgoing payments to hosts.
    - Includes amount, host, payout method, status, gateway, timestamps.
- `refunds`:
    - Records amounts refunded to guests, with links to payment and booking and reason codes.
- `transactions` or `ledger_entries`:
    - This is an internal accounting table designed in a double-entry fashion.
    - Every financial movement results in at least two entries: debit and credit.

    - Accounts might include platform_wallet, guest_wallet, host_wallet, gateway_receivable, gateway_fees, etc.
    - This can start simple (just track platform vs host vs guest) and later be extended to full accounting where you can generate financial statements.

Support and disputes

- `support_tickets`:
    - Represents any support case, such as “property not as described”, “host no-show”, “payment not reflecting”, “damage dispute”, etc.
- `disputes`:
    - Specific entity for damage or refund disputes tied to a booking.
    - Includes claimant, respondent, evidence, status, resolution, and any adjustment made to payouts/deposits.

KYC and verification

- `kyc_documents`:
    - Records the type of document (NIDA, passport, business registration, tax ID), status (pending, approved, rejected), file reference, and notes.
- `property_verifications`:
    - Record of property inspections done by your or a partner’s City Ops team.

Notifications and events

- `notifications`:
    - Tracks push, SMS, and email notifications sent to a user, status, type, and related booking or property.
- `system_events`:
    - Optional but useful table that logs key domain events like booking_created, booking_paid, booking_cancelled, payout_sent, etc., which can later drive CRM automation and analytics.

Access control and smart lock preparation

- `access_control_devices`:
    - In the future, each property can have associated devices like smart locks.
    - This table stores provider name, device id, configuration, and status.
- `access_codes`:
    - Codes generated per booking or per stay, tying a property and a guest to a temporary code.
    - For now, this can be empty or used only with manual codes; later, integration with a real smart lock API can fill this automatically.

Loyalty and marketing preparation

- `loyalty_accounts`:
    - One per guest, tracks total points, tier, etc.
- `loyalty_transactions`:
    - Accrual and redemption of loyalty points.
- `promo_codes` and `promo_redemptions`:
    - For discounts and marketing campaigns.

Multi-country preparation

Even though you start in Tanzania only, many tables should include a `country_code` and/or `currency_code` field, plus time zone where necessary. You will store values like “TZ” and “TZS” now, but this prevents painful migrations when you expand.

---

5. Payment architecture and Tanzanian gateways

You cannot depend on Stripe in Tanzania. Your primary reality is mobile money and local payment aggregators like AzamPay and Selcom. You design the backend so that the app does not know or care which gateway is used; it just knows the flow: initiate payment, get payment URL or instructions, check status, then mark booking as paid.

Backend has a dedicated payment service layer inside FastAPI. Conceptually:

- The app calls your backend to initiate a payment for a specific booking.
- The backend:
    - Checks booking data and amount.
    - Creates a `payments` row with status “initiated”.
    - Calls the selected gateway’s API (AzamPay or Selcom) using its SDK or HTTP.
    - Stores the gateway reference in the payment record.
    - Returns to the app whatever is needed: a redirect URL, payment token, instructions.
- The guest completes the payment using their mobile money or card through the gateway.
- The gateway calls your webhook URL when payment is successful or fails.
- Your backend validates the webhook, updates the `payments` row, and if payment is successful:
    - Updates the booking status to “confirmed”.
    - Creates accounting entries in the ledger.
    - Schedules payouts to the host.

To keep things clean:

- You never write AzamPay-specific logic into booking code directly. The booking service calls a generic payment service interface, which internally dispatches to the correct gateway implementation.
- Every payment record knows:

- Which gateway was used.
    - A unique idempotency key so that multiple webhooks do not double-process the same payment.
- There is a reconciliation process:
    - A scheduled job that reads recent payments and cross-checks them with the gateway API to detect any inconsistencies (for example, webhooks that never arrived).

For payouts:

- You create payouts either automatically after the stay plus buffer period or in batch.
- Each payout is tied to host, bookings covered, and payment gateway or bank route.
- Payouts ideally go through the same or another aggregator that can handle mobile wallet or bank transfers.

This design also makes it easy to add:

- Card payments via gateways that support Visa/Mastercard later.
- Split payments at checkout: your ledger knows how to allocate money between platform and host, and the gateway receives a single debit but you handle accounting internally.

---

6. Booking lifecycle and business rules

A booking moves through a clear lifecycle:

1. Guest searches and chooses dates and a property.
2. Backend checks availability and valid pricing and returns a quote.
3. Guest confirms they want to book; a tentative booking record is created with status “pending” or “awaiting_payment”.
4. Guest pays through integrated gateway.
5. On successful payment, booking is marked as “confirmed”.
6. A few days before check-in, the system sends reminders and access instructions.
7. Check-in:
    - Host or property manager confirms the guest has arrived, or the guest confirms through the app.
    - Future: smart lock codes become active at check-in.
8. During stay, the guest can contact support or request extension.
9. Check-out:
    - Host inspects the property.
    - If no issues, deposit is released after a fixed window (for example, 24 hours).
    - If there are issues, host files a dispute tied to the booking.
10. After resolution, both guest and host can leave reviews.
11. Booking moves to “completed”.

Cancellation rules:

- Each property has a cancellation policy (flexible, moderate, strict).
- The business logic determines how much of the paid amount is refundable based on:
    - Time of cancellation relative to check-in.
    - Policy rules.
    - Special cases (force majeure).
- Backend calculates the refund, creates a `refunds` record, and triggers the appropriate gateway action.
- Ledger entries reflect movement of refund amounts.

All of this should be implemented as pure business logic in FastAPI services, separate from the HTTP routes. That makes it easier to test and evolve.

---

7. Host onboarding and property verification

You do not want a random host signing up, uploading low-quality photos and flipping scams through your platform. So host onboarding is a structured flow.

Host side:

1. The user signs up via Clerk, then switches to host role.
2. Host profile is created and marked as "unverified".
3. Host submits required information:
    - Individual or business.
    - Personal or director details.
    - Mobile number and payout destination.
    - Identification documents (NIDA, passport, business registrations, etc.).
4. KYC documents are uploaded; references go into `kyc_documents`.

Admin side:

1. Admin reviews KYC documents in the admin panel.
2. Approves or rejects host profile, sets `verification_status`.
3. Optionally, requests more information.

Property verification:

1. Host creates a new property and fills in address, amenities, rules, and initial photos.
2. Property status starts as "draft".
3. Once all required fields are filled and at least minimal photos are uploaded, status can move to "pending_verification".
4. A City Ops member or partner receives a verification task:

    - They visit the property, check basic safety, confirm address, and take their own photos.
    - They upload inspection notes and verified photos.
5. Admin approves and sets property to "verified"; these properties can show a badge in the app.

This pipeline also provides the hooks to later integrate more advanced verification like remote video inspections or integration with property managers.

---

8. Background jobs, scheduled tasks and "edge" behaviour

Not everything can run in the HTTP request cycle. For example:

- Sending emails and SMS messages.
- Processing webhooks and long-running gateway requests.
- Generating reports.
- Running daily reconciliation.
- Expiring pending bookings that were never paid.
- Automatically adjusting availability if bookings expire.

For these jobs you run a worker process that pulls tasks from a queue. Minimal setup:

- Redis for task queue.
- RQ or Celery or a similar lightweight task queue library.

The FastAPI app puts jobs on the queue; the worker consumes them.

Examples of tasks:

- send_notification(user_id, type, payload)
- process_payment_webhook(payment_id, raw_payload)
- run_daily_reconciliation()
- expire_unpaid_bookings()

Later, if you want some functions closer to "edge" style (for example, Cloudflare Workers), you can offload static operations like image resizing or very simple health checks. But the core should remain under your control on the VPS for now.

---

9. Mobile application structure with Expo and Atomic Design

The mobile app is your main interface. Expo with React Native gives you cross-platform iOS and Android with a single codebase.

Use Atomic Design principles to stop the UI from turning into spaghetti:

- Atoms are things like buttons, text inputs, icons, typography components, spacing primitives.
- Molecules are combinations of atoms, like an input field with label and error, or a property card with image, name, and price.
- Organisms are larger structures like a booking summary card, property details header, or search filter panel.
- Pages (screens) are built from these organisms.

Screens might include:

- Onboarding and authentication screens (connected to Clerk).
- Home/search screen with search bar, filters and property list.
- Property detail screen with photos, description, map, reviews, and call to action.
- Booking screen with date selection, pricing breakdown, and confirmation.
- Payment screen where the app redirects or presents gateway flow.
- Guest profile and bookings history.
- Host dashboard: property list, earnings, booking management.
- Property edit and add screens.
- Support and help center screen.
- Review submission screens.

Cross-cutting concerns:

- Global state for user info, auth token, and minimal cached data. A simple state management library like Zustand or Redux can be used.
- API client module that handles all communication with FastAPI, including attaching auth token, handling errors, and retrying certain calls.
- A small offline strategy: at least cache profile, saved searches, and last viewed properties for when the network is weak.
- Consistent loading and error states so the app feels responsive even on unstable connections.

Performance considerations:

- Always paginate lists of properties.
- Avoid heavy layouts on low-end devices.
- Lazy-load images and use thumbnails where appropriate.
- Avoid unnecessary re-renders by splitting components and using memoisation where needed.

---

10. Admin panel and internal tools

You need internal control to manage the platform, even if at first you are the only admin.

Admin panel responsibilities:

- See list of users, filter, search, and view details.
- See host profiles and KYC status; approve or reject hosts.
- See properties, verification status, and content. Approve, reject, or hide them.
- View bookings and their status; manually override if needed.
- View payments, payouts, and refunds; trigger manual payouts or adjustments.
- Handle support tickets and disputes; attach notes and resolutions.
- See system metrics: number of active listings, bookings per day, revenue, cancellations, disputes.

Implementation options:

- A simple React web app that consumes the same backend API with admin endpoints.
- Or a minimal admin front end built using FastAPI's templating at first, then replaced later.

The important part is that admin actions are logged, auditable, and protected by role-based access control.

---

11. Logging, monitoring, and observability

To keep this thing alive in production, you need to know what is happening. At minimum:

- Structured logs from FastAPI: each request logs endpoint, user (if authenticated), response time, and status code.
- Error logging: any exception gets logged with stack trace and context, and ideally reported to an external error tracker.
- Separate logs for background jobs and workers.
- Simple metrics: requests per minute, error rate, average response time, queue length, delayed jobs.

This does not need to be complex at launch, but do not ignore it. Even a simple setup with a log aggregation service or just structured logs on disk that you can tail and search is better than nothing.

---

12. Security and data protection

There are a few non-negotiables:

- All communication between app and backend must use HTTPS.
- The backend must validate all Clerk tokens properly and never trust user ids from the client blindly.
- Role-based access control enforced on every endpoint: guests cannot access host-only endpoints, etc.
- File uploads for images must go through signed upload flows; you should not allow arbitrary file types or direct backend file storage without validation.
- Webhook endpoints from payment gateways must implement signature verification or IP allow-listing where possible.
- Sensitive configuration (gateway keys, Clerk keys, DB passwords) must never be hard-coded. They live in environment variables or a proper secret manager.
- For ID documents and any KYC files, either:
  - Store only minimal references and keep full files offsite in a secure location, or
  - At least encrypt them at rest and strictly limit who can access them.

---

13. Preparing cleanly for future features without overbuilding

The future add-ons you named are serious: smart locks, card payments, AI pricing, CRM automation, community features, multi-country support, insurance, loyalty, split payments, full accounting.

The way you prepare for them now is through design decisions, not by building them early. A few key patterns:

- Never hard-code a single payment gateway. Always go through a payment service interface.
- Always include country, currency, and time zone in relevant tables.
- Do not model money as random floats everywhere; use integer minor units (cents) and one central type to represent amounts and currencies.
- Use event logging for key business actions so that later, CRM workflows and AI models can replay behaviour.
- Keep property pricing and availability logic in dedicated services, not scattered across routes. When you introduce an AI pricing engine, it will plug into those services.
- Keep your accounting records in a form that naturally extends to double-entry: every money movement is two entries with consistent totals.

This way, when you are ready to for example turn on dynamic AI-based pricing, you only need to:

- Build a service that reads historical bookings and events.

- Computes suggested prices.
- Writes them back to pricing tables.
- Exposes them through existing APIs.

Without touching the booking or payment core.

---

14. Development workflow and using AI tools effectively

Given you plan to use Cursor, Claude and similar tools, your repository should be easy for them to reason about:

- Code is organised by domain area (auth, users, bookings, payments, properties) rather than random folders.
- Each module has clear interfaces and docstrings that describe what it does.
- Database schema is documented and consistent with migrations.
- Business logic is kept in services, which means AI can generate tests and changes in a focused way.

Typical order of implementation:

1. Data models and migrations.
2. Core auth integration and mapping between Clerk and your `users` table.
3. Basic CRUD endpoints for properties and profiles.
4. Booking creation and pricing calculation.
5. Payment integration and webhooks.
6. Booking confirmation and cancellation logic.
7. Payouts and transaction ledger.
8. Reviews and support tickets.
9. Notifications and background jobs.
10. Admin endpoints.