PURSUIT

Local Event Discovery · Nairobi, Kenya

**A full-stack mobile app built solo,
from schema design to shipped UI.**

React Native | Expo | Django | GraphQL | PostgreSQL | TypeScript | Redis

About this project

Pursuit is a React Native mobile app for discovering and planning local events in Nairobi, Kenya.

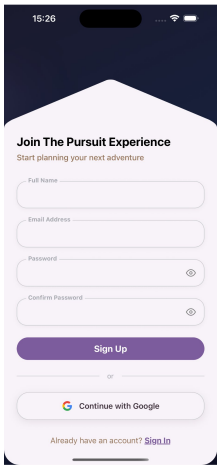
Built entirely solo, covering product design, API architecture, database schema, and mobile UI. The backend runs on live staging infrastructure.

Operated under Pursuit HQ, a registered sole proprietorship.

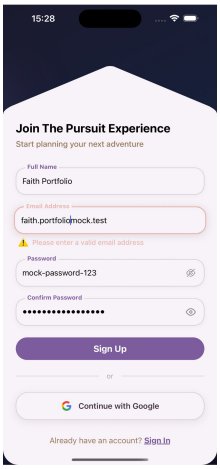
Full-Stack	Live	8 Pages	Real Data
Solo build	Staging environment	Full case study	Nairobi events via API

Authentication

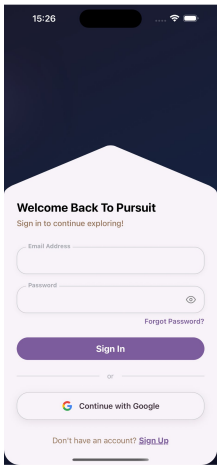
Sign up · Sign in · Form validation · Google OAuth



Sign up — empty state



Sign up — validation error



Sign in screen

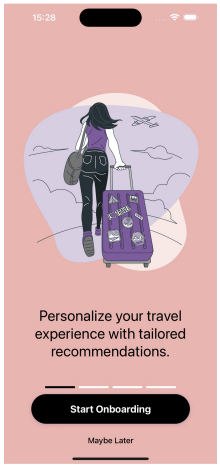
Technical highlights

- Custom form components with floating labels that persist above the field on focus, so users never lose field context mid-form.
- Client-side validation runs before any network call: email format is checked in real time and a password visibility toggle is available on all password fields.
- Google OAuth is handled via expo-auth-session, which triggers the native iOS permission dialog. The id_token is then exchanged server-side with the Django backend.
- On login, the backend returns a JWT access token, refresh token, and session token. These are stored securely on device and sent as the Authorization header on all subsequent requests.
- Two separate GraphQL mutations handle sign-out: signOut revokes only the current device session, while signOutAll revokes all active sessions. This distinction is a deliberate security design for account compromise scenarios.

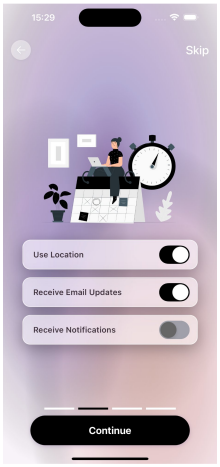
expo-auth-session **JWT tokens** **Django auth** **Client-side validation** **Google OAuth** **Session management**

Onboarding Flow

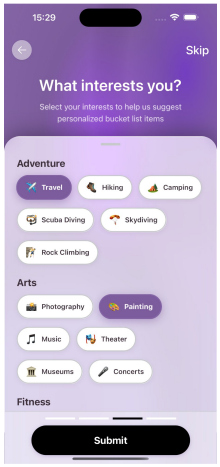
Splash screen · Permissions and preferences · Interest selection



Welcome splash



Permissions and preferences



Interest selection

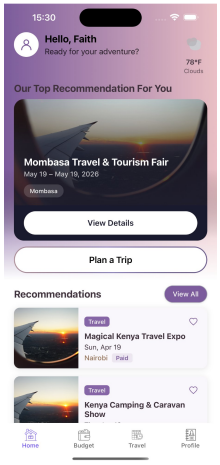
Technical highlights

- Multi-step wizard with progress dot indicators and a persistent Skip button on every step. User agency is respected throughout; completion is never forced.
- Location access, email updates, and push notification preferences are captured as toggles and stored against the UserProfile model on the Django backend.
- Interest selection uses a pill-chip grid inside a bottom sheet. It supports multi-select, and selected chips fill with brand purple for clear visual feedback.
- Selected interests are saved as a ManyToMany relation on UserProfile.interests and passed into the GraphQL recommendations resolver on every home screen load.
- expo-location and expo-notifications handle native permission requests. If either is denied, the app falls back gracefully: hasLocation returns false and only city-level results are shown.

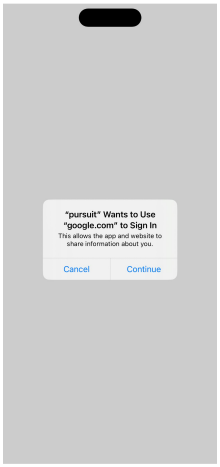
Expo Router **expo-location** **expo-notifications** **ManyToMany interests** **Personalisation layer** **Bottom sheet UI**

Home Screen

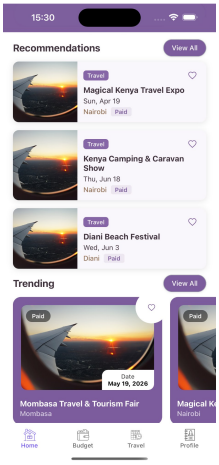
Personalised feed · Upcoming trip · Weather widget · Recommendations



Top recommendation hero



Upcoming trip and events



Recommendations list

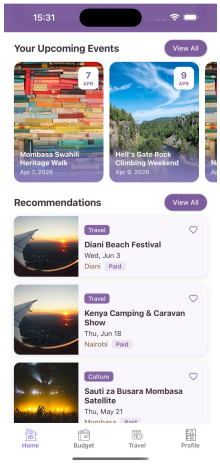
Technical highlights

- The home feed is driven by a single GraphQL getHome query. Greeting, weather, insights, recommendations, and upcoming events are all fetched in one round-trip.
- A weather widget calls an external weather API on load and surfaces the current temperature and condition in the screen header.
- The upcoming trip hero card displays trip metadata including title, date range, number of nights, and event count, sourced from the trips and events relational model.
- The Upcoming Events section pulls user-saved events ordered by date. Each card shows a date badge, cover image, event title, and location.
- Recommendations are ranked by interest category match. The resolver cross-references UserProfile.interests with event categories stored in PostgreSQL.
- Bottom navigation uses four tabs: Home, Budget, Travel, and Profile. The layout is built with Expo Router and uses icons with labels throughout for accessibility.

GraphQL getHome **Weather API** **Trips data model** **Interest matching** **Expo Router tabs** **Personalised feed**

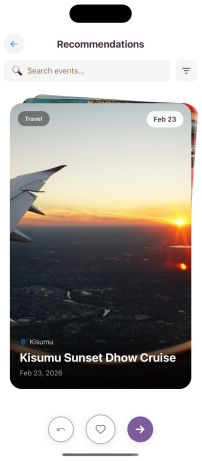
Events: Browse and Discover

Swipe cards · List view · Search and filter · Personalised recommendations



Swipeable event cards

Upcoming events list



Personalised recommendations

Technical highlights

- Events can be browsed via a swipeable card stack. Swiping right saves an event; swiping left skips it. Animated gesture handling is built directly in React Native.
- List views include a search bar with debouncing. The query is only sent to the GraphQL backend after the user pauses typing, which avoids unnecessary API calls on every keystroke.
- Each event card shows a cover image, category badge, title, date, location, paid or free status, and a save toggle.
- The recommendations resolver filters events by the user's interest categories and location radius. The search radius in kilometres is stored on the UserProfile model.
- The events data model in PostgreSQL supports category, location, date range, pricing, and trip association.
- All event data is served through GraphQL. The resolver uses select_related to fetch nested event, category, and location data in a single query, preventing N+1 issues.

Gesture handling **Debounced search** **GraphQL resolvers** **Category filtering** **select_related** **PostgreSQL**

Stack and Architecture

Full-stack overview: frontend, backend, and infrastructure

Frontend

- React Native + Expo
- TypeScript throughout
- Expo Router (file-based nav)
- Apollo Client for GraphQL
- expo-auth-session (OAuth)
- expo-location and notifications
- Custom component library
- Gesture handling for swipe cards

Backend

- Django 5.0 (Python)
- Graphene-Django (GraphQL)
- Django REST Framework
- JWT and session tokens
- PostgreSQL 17
- Redis (cache and task broker)
- Celery for async tasks
- Environment-split settings

Infrastructure

- Render.com with render.yaml IaC
- Docker containerised
- Staging environment (auto-deploy)
- Isolated staging database
- ipAllowList: [] (internal only)
- Health check endpoint
- Secrets auto-generated at deploy

Data flow

React Native (Apollo Client) sends queries to the GraphQL API (Graphene-Django), which reads from and writes to PostgreSQL 17.

Auth: a JWT access token, refresh token, and session token are issued on login. Tokens are stored on device and sent as the Authorization header on every request.

Google OAuth: expo-auth-session triggers the native iOS dialog. The id_token is exchanged server-side and Django issues its own JWT pair.

What this demonstrates as a contractor

End-to-end ownership: every layer was designed and built solo, from the data model and API through to the mobile UI.

Sound engineering practices: JWT authentication, GraphQL schema design, Redis caching, environment isolation, and infrastructure as code.

React Native depth: custom components, gesture handling, native permission flows, and Expo Router navigation.

Backend maturity: Celery for async tasks, Redis cache-aside pattern, Docker containerisation, and health check endpoints.

UI/UX Design Decisions

Intentional choices behind every screen, not just implementation

Floating label inputs

Standard placeholders disappear on focus, leaving users unsure which field they are editing.

Floating labels persist above the input when focused or filled. This is especially important on multi-field auth forms where losing context causes mistakes.

Applied consistently across both sign up and sign in.

Home screen hierarchy

Most apps open with generic browsable content. Pursuit opens with the user's upcoming trip, surfacing personal commitment before discovery.

This reduces scroll distance to the content that matters most. Recommendations and trending events follow below the fold.

A deliberate information architecture decision, not a default layout.

Swipe cards for events

List rows require reading each item in sequence. Swipe cards surface one event at a time with a full cover image and all key details visible.

This makes save-or-skip decisions faster and more tactile. Right swipe saves; left swipe skips.

Well suited to discovery-mode browsing, where users are exploring rather than searching for something specific.

Interest chips instead of dropdowns

Dropdowns on mobile require two taps and hide all options until opened. Chips are rendered as a pill grid in a bottom sheet, making all options visible at once.

Chips support multi-select and use brand purple fill on selection, so the user's choices are scannable at a glance.

Faster and clearer than a dropdown or checkbox list for this use case.

Progress dots on onboarding

Labelling steps as "2 of 4" frames onboarding as a task list and creates pressure to rush through.

Dots give a spatial sense of progress without counting anxiety. The Skip button is always visible, keeping the user in control at every step.

This approach reduces abandonment by removing friction and time pressure from the flow.

Four-tab bottom navigation

More than four tabs creates thumb-reach problems on large phones and risks burying core features.

Home, Budget, Travel, and Profile cover all primary user actions without overflow menus or hidden navigation.

Icons are paired with labels throughout, not used alone, for full accessibility clarity.

Consistent design system

Brand purple (#7B5EA7) is used for all interactive elements, calls to action, selected states, and accent colours across every screen.

Two background modes are used: dark navy for immersive screens such as auth and onboarding, and off-white for content screens such as home and events.

Card components are fully reused across home, recommendations, and event list views with the same padding, border radius, and image aspect ratio.

New screens can be composed from existing components with no one-off styling required.

Backend and System Design Decisions

Architecture reasoning: the why behind every major choice

Why GraphQL over REST

The home screen requires greeting text, weather data, insights, recommendations, and upcoming events all in a single render. With a REST API, that would mean five or more separate calls, each adding round-trip latency on a mobile connection.

The GraphQL getHome query fetches all of this in one request, returning exactly the fields the client needs with no over-fetching and no waterfall loading. This specific data shape was the decisive reason for choosing GraphQL.

Location syncing

The app requests foreground location permission via expo-location. When granted, the device coordinates are captured and sent to /api/user-preferences/, where they are stored as homeLatitude and homeLongitude on the UserProfile model.

UserProfile also stores a searchRadiusKm field, defaulting to 50km and set during onboarding. The recommendations resolver uses this radius to filter nearby events using a SQL distance calculation. PostGIS is not required at the current data scale.

If location permission is denied, hasLocation returns false and the home screen falls back to city-level event results with no personalised radius applied.

Caching with Redis

The getHome resolver is the most expensive query in the system. It aggregates weather data, user insights, and event lists on every app open.

A cache-aside pattern is used: Redis is checked first, and the database is only queried on a cache miss. Results are written back with a TTL. Weather data uses a 30-minute TTL. Recommendations use a one-hour TTL. The full home aggregate uses 15 minutes.

Cache keys include the user ID to prevent any possibility of cross-user data being returned. Redis also acts as the Celery task broker for async notification delivery.

Infrastructure as code: render.yaml

The staging environment auto-deploys on every push to main, enabling fast iteration. This is a deliberate choice to keep staging ahead of any manual process.

The staging database is isolated with ipAllowList: [] set to internal connections only. A migration run against staging cannot affect any other environment.

SECRET_KEY and JWT_SECRET_KEY are set using generateValue: true. Render generates cryptographically secure values at deploy time. No secrets are stored in the codebase or committed to version control.

GraphQL | Redis cache-aside | expo-location | render.yaml IaC | JWT | Celery | PostgreSQL 17 | Docker | select_related