Njangui

Technical Documentation and Developer Guide (English)

Njangui Engineering Team

2026-04-06

Abstract

This technical guide documents the current Njangui implementation with a focus on architecture, setup, workflow invariants, compatibility strategy, quality engineering, and release governance.

Contents

1 Technical Overview

1.1 Purpose

This document is the engineering reference for Njangui.

It explains: - runtime architecture, - domain boundaries, - workflow invariants, - data persistence strategy, - quality and release practices.

1.2 Product baseline

At this revision: - versionName: 1.1.0 - versionCode: 4 - code name: AJEMI−UDs

1.3 Engineering goals

1. Keep business logic in-app (rich client model).
2. Keep backend persistence replaceable (Firebase is current storage, not the domain owner).
3. Preserve schema compatibility across environments and historical documents.
4. Keep role/rights checks explicit and centralized.
5. Keep accounting and reporting auditable.

1.4 Documentation map

- 02−architecture.md: components, data model, permissions
- 03−development−setup.md: local setup and commands
- 04−core−workflows.md: critical lifecycle flows
- 05−testing−and−quality.md: test strategy and quality gates
- 06−contribution−guide.md: contribution rules and review checklist
- 07−release−baseline−and−changelog.md: baseline + release policy
- 08−rule−to−code−traceability.md: business rule to implementation mapping

2 Architecture

2.1 1. Technology stack

- Kotlin + Android Views/XML + ViewBinding
- Gradle Wrapper 9.3.1
- Android Gradle Plugin 9.1.0
- Firebase Auth + Firestore
- WorkManager for periodic sync and notifications
- Glide for profile image loading

2.2 2. High-level architecture

Njangui uses a client-centric layered architecture:

- **View layer** (views/)
  - Activities, dialogs, user interactions, UI state.
- **Port contracts** (providers/ports/)
  - Business-facing interfaces.
- **Port implementations** (providers/ports/impl/v1/)
  - Firestore/Auth implementations and domain workflows.
- **Domain models** (providers/models/)
  - Serializable business structures.
- **Utilities** (utils /)
  - permissions, validation, formatting, notification processors.
- **Background runtime** (workers/, receivers /)
  - periodic lifecycle sync and notification delivery.

The Providers class is used as a lightweight service locator with lazy setup.

2.3 3. Runtime components

2.3.1 `NjanguiApp`

- applies persisted theme mode,
- schedules notification worker:
  - periodic pull every 15 minutes,
  - immediate one-shot run on startup.

2.3.2 `BootReceiver`

- re-schedules worker after reboot/update.

2.3.3 `NotificationPullWorker`

During each run, it executes: 1. tontine lifecycle sync, 2. governance lifecycle sync, 3. penalties lifecycle sync, 4. caisses lifecycle sync, 5. events lifecycle sync, 6. unread notification pull and local delivery.

2.4 4. Data persistence model

Primary Firestore collections:

- users
- notifications
- tontine−groups−basic
- tontine−groups−caisses
- tontine−groups−elections
- tontine−groups−events

- tontine−groups−tontines
- tontine−groups−penalties

Local settings (SharedPreferences via AppSettingsPortImpl): - language, - theme mode, - default active group, - last pulled notification id.

2.5 5. Permission architecture

Rights enum: - MANAGE_MEMBERS - MANAGE_TONTINES - MANAGE_CAISSES - MANAGE_PENALTIES - MANAGE_EVENTS

Permission decisions are centralized in PermissionsChecker and depend on: - active membership, - active staff mandate, - staff-right ownership, - governance fallback windows for creator/admins.

2.6 6. Key domain structures

- TontineGroupBasicModel: group identity, members, admins, staff
- TontineModel: participants, ranking, rounds, lifecycle flags
- RankingPreConditionsModel: strategy, fixed positions, ranking mode, deadline
- CaisseModel: individual/shared balances, transactions, low-balance policy
- GroupEventModel / MainLeveeModel: event and main levee workflows
- PenaltyModel: disciplinary lifecycle with recoveries
- FinancialReportModel: aggregated finance output

2.7 7. Compatibility strategy

Because staging carries real usage and historical schema variations, implementations follow tolerant read/write behavior:

- nullable fields and defaults in models,
- defensive parsing for dates/enums,
- no-arg constructors preserved for Firestore model mapping,
- reconstruction of missing notification document fields where needed,
- migration-safe behavior over destructive rewrites.

2.8 8. Architectural invariants

1. Business decisions are not encoded in UI widgets only.
2. Rights checks happen before every sensitive mutation.
3. Accounting mutations generate transaction traces.
4. Reports are derived from filtered datasets, not hidden global state.
5. Notification delivery must avoid duplicate and wrong-recipient exposure.

3 Development Setup

3.1 1. Prerequisites

- Android Studio (latest stable recommended)
- Java 17+
- Android SDK (API 35 recommended)
- Git
- Firebase project access (Auth + Firestore)

3.2 2. Clone repository

```
git clone <repository-url>
cd NjanguiApp
```

3.3 3. Local configuration

Create/update config.properties at repository root:

```
DEV_DB_SUFFIX=-test
PROD_DB_SUFFIX=-staging
PROD_KEYSTORE_FILE=KeyStore.jks
PROD_KEYSTORE_PASSWORD=<password>
PROD_KEYSTORE_ALIAS=<alias>
PROD_KEYSTORE_KEY_PASSWORD=<password>
CODE_NAME=AJEMI-UDs
```

3.4 4. Firebase setup

1. Enable Email/Password in Firebase Auth.
2. Enable Firestore.
3. Register Android package miu.sts.app.njangui.
4. Place google-services.json in Njangui/google-services.json.

3.5 5. Build and test commands

Build debug APK:

```
./gradlew :Njangui:assembleDebug
```

Build release APK:

```
./gradlew :Njangui:assembleRelease
```

Unit tests:

```
./gradlew :Njangui:testDebugUnitTest
```

Instrumented tests (device/emulator connected):

```
./gradlew :Njangui:connectedDebugAndroidTest
```

3.6 6. Documentation build

```
cd docs
make pdf
make html
```

Output root: docs/build/

3.7 7. Environment notes

- Staging is actively used by a real group; avoid breaking schema compatibility.
- Never assume all documents contain all fields.
- Add backward-compatible defaults for new fields.

3.8 8. Troubleshooting

3.8.1 Android SDK not selected

If build config reports SDK selection issues, open Android Studio SDK Manager and ensure target SDK/platform tools are installed and selected.

3.8.2 Firestore deserialization failures in release

Typical root causes: - missing no-arg constructor, - R8/ProGuard stripping required classes/constructors.

Mitigation: - keep model classes Firestore-friendly, - add keep rules for reflective/deserialized classes.

3.8.3 Worker not delivering notifications

Check: - app has notification permission, - user is authenticated, - WorkManager scheduled jobs exist, - channel njangui_notification_channel created.

4 Core Workflows

This section documents the most critical domain flows and their invariants.

4.1 Visual map (Mermaid)

```
flowchart TD
    A[User Action] --> B{Permission Check}
    B -->|Denied| C[UI Feedback + Stop]
    B -->|Granted| D[Port Impl Mutation]
    D --> E[Firestore Write]
    E --> F[Lifecycle/Derived Effects]
    F --> G[Notification Persisted]
    G --> H[Worker Pull]
    H --> I[Local Device Notification]
    F --> J[Report Aggregation Inputs]
```

4.2 1. Authentication and account eligibility

```
sequenceDiagram
    participant U as User
    participant L as LoginActivity
    participant A as AuthPort
    participant F as FirebaseAuth
    participant D as UserDocument

    U->>L: Submit credentials
    L->>A: login(email, password)
    A->>F: signIn
    F-->>A: auth success/failure
    A->>D: fetch business user state
    D-->>A: account status
    A-->>L: allow or deny
```

Invariants: - Firebase authentication success is necessary but not sufficient. - Business account state must also allow access.

4.3 2. Group context synchronization

```
ProtectedBaseActivity
  -> resolve default group
  -> verify active membership
  -> update local settings
  -> populate drawer switcher and screen
```

Invariants: - all module operations use the active group context, - stale group context must be corrected before mutating data.

4.4 3. Permission resolution

Centralized in PermissionsChecker.

Decision factors: - active membership, - active staff mandate, - delegated staff rights, - super-admin fallback windows after mandate expiry.

Invariants: - no sensitive mutation without explicit permission check, - MANAGE_MEMBERS has dedicated fallback logic unlike other rights.

4.5 4. Governance lifecycle

4.5.1 Election

```
stateDiagram-v2
    [*] --> DraftElection
    DraftElection --> VotingOpen: create + notify voters
    VotingOpen --> Consolidation: all votes or deadline reached
    Consolidation --> NewStaffActive: apply results
    NewStaffActive --> [*]
```

4.5.2 Staff replacement

```
Request replacement
  -> collect approvals from eligible staff
  -> apply replacement once threshold met
  -> notify group
```

Invariants: - one active mandate window at a time, - replacements follow approval rules before effective application.

4.6 5. Tontine ranking lifecycle

4.6.1 Configuration

RankingPreConditions includes: - multipleNamesStrategy: NONE or BALANCED - fixed positionning - rankingType: PARTICIPANT_SELECTION or RANDOM_ALGORITHM - participant-selection deadline

4.6.2 Execution and closure

```
flowchart LR
    A[Configure preconditions] --> B[Participant picks or algorithm picks]
    B --> C{Missing slots?}
    C -->|Yes| D[Auto-complete on closure/deadline]
    C -->|No| E[Finalize ranking]
    D --> E
    E --> F[Notify ranking changes]
```

Invariants: - rule updates can invalidate existing picks, - fixed positions have priority, - closure may auto-fill missing picks.

4.7 6. Round lifecycle (tontine)

```
round starts -> notify all participants
  -> contributions captured (draft/final)
  -> outflows specified for beneficiaries
  -> close round
  -> generate penalties if needed
  -> archive successful round
  -> notify completion
```

Invariants: - round closure requires MANAGE_CAISSES, - contribution and outflow payloads must be coherent, - ending condition is automatic when all rounds are completed.

4.8 7. Caisse accounting workflow

4.8.1 Core model

All monetary effects are represented through transactions: - direction: credit/debit, - typed references (referenceType, referenceId), - optional counterparty and note metadata.

4.8.2 Shared caisse distribution

```
distribute shared caisse amount
  -> validate shares (amount/percent)
  -> create debit from source
  -> create credits/outflows by destination
  -> persist and report
```

Invariants: - accounting operations must be balanced by design, - deletion of caisse must remove its accounting transactions.

4.9 8. Events and main levees

4.9.1 Events

Supports: - individual contribution mode with caisse priorities, - collective contribution mode from shared caisses, - optional external beneficiaries at creation, completed at closure if needed.

4.9.2 Main levees

Supports: - compulsory/optional campaigns, - progressive contribution capture, - closure with sanction generation on rule violations.

Invariants: - outflow closure checks required amounts and eligible caisse types, - rollback paths exist for incorrect outflow definitions.

4.10 9. Penalties lifecycle

```
stateDiagram-v2
    [*] --> PendingValidation
    PendingValidation --> Validated
    Validated --> Paid
    Paid --> PaidPendingRecovery
    PaidPendingRecovery --> RecoveryClosed
    PendingValidation --> Cancelled
    Validated --> Cancelled
```

Invariants: - penalties may include amount and text fines, - amount fines require explicit recovery destination and reason where applicable, - disciplinary and financial reports must include recovery effects.

4.11 10. Notifications lifecycle

Notification flow:

```
sequenceDiagram
    participant B as Business Mutation
    participant N as Notifications Doc
    participant W as NotificationPullWorker
    participant P as NotificationTypesAndProcessors
    participant D as Device NotificationManager

    B->>N: append notification payload
    W->>N: pull unread notifications
    W->>P: process by type
    P->>D: build + display local notification
```

Invariants: - recipient targeting must be strict, - duplicate delivery must be prevented via deliveredTo and local max-id tracking, - missing legacy fields should be reconstructed when possible.

4.12 11. Reporting workflow

4.12.1 Finance reports

Sources include: - rounds, - caisses, - events/main levees, - penalties and recoveries, - internal/external beneficiary flows.

4.12.2 Penalty reports

Must expose: - state transitions, - paid vs unpaid, - context (group/tontine/round), - due dates and closures.

Invariant: - exports are generated from the currently filtered dataset only.

5 Testing and Quality

5.1 1. Quality objectives

Njangui quality is defined by: - business-rule correctness, - permission safety, - accounting consistency, - schema compatibility, - stable notification delivery.

5.2 2. Automated tests (current baseline)

Representative unit test coverage includes: - validators and string helpers, - permissions and governance logic, - identity rendering helpers, - caisse accounting invariants.

Known classes: - FieldValidatorTest - StringUtilsTest - PermissionsCheckerTest - UserIdentityUtilsTest - CaisseAccountingUtilsTest

Run tests:

```
./gradlew :Njangui:testDebugUnitTest
```

5.3 3. Recommended test matrix for new features

1. **Rights matrix**
   - member vs staff vs admin behavior,
   - mandate active/expired cases.
2. **Lifecycle matrix**
   - creation/update/closure paths,
   - rollback and retry behaviors.
3. **Schema matrix**
   - complete documents,
   - partially missing fields,
   - legacy values and enum fallbacks.
4. **Accounting matrix**
   - debit/credit balance effects,
   - shared vs individual caisse,
   - recovery destination scenarios.
5. **Notification matrix**
   - correct recipient targeting,
   - duplicate prevention,
   - background + foreground behavior.

5.4 4. Manual QA high-risk scenarios

- ranking rule update after participant picks,
- round closure with partial contributions and redirection,
- event collective outflow with shared caisse constraints,
- penalty paid -> recovery closure workflow,
- language/theme switch while on Settings and other active screens,
- report filters followed by export.

5.5 5. Logging and diagnostics

When diagnosing: - capture stack trace + timestamp, - include user role and active group, - include workflow step and expected state transition, - include relevant Firestore document ids.

5.6 6. Documentation quality gate

Docs pipeline validates: - Pandoc + TeX availability, - required LaTeX style packages, - PDF and HTML generation, - publishable artifact under docs/build.

5.7 7. Definition of done (quality)

A feature is done when: 1. behavior is correct across rights/lifecycle constraints, 2. data compatibility is preserved, 3. regression-prone paths are tested, 4. logs/alerts are actionable, 5. documentation and changelog are updated.

6 Contribution Guide

6.1 1. Core engineering rules

1. Keep business rules in ports/utils, not in ad-hoc UI branches.
2. Gate all sensitive mutations behind centralized permission checks.
3. Preserve backward compatibility for Firestore documents.
4. Keep financial mutations auditable through transactions.
5. Prefer explicit status transitions over implicit booleans.

6.2 2. Coding conventions

- Kotlin naming conventions (PascalCase for classes, camelCase for members)
- feature-oriented packages
- null-safe parsing for external/persisted data
- defensive date and enum parsing
- user-safe error feedback from view layer

6.3 3. Branch and pull-request workflow

Recommended flow:

1. create one branch per scope,
2. implement + test + document,
3. open PR with clear impact summary,
4. request review,
5. merge after quality checks pass.

Commit style:

```
type(scope): short summary
```

Examples: - feat(events): add collective outflow closure safeguards - fix(notifications): prevent duplicate deli
- docs(functional-fr): detail penalty recovery lifecycle

6.4 4. Schema evolution policy

When adding/changing persisted fields:

1. add nullable field/default in model,
2. keep tolerant mappers for missing values,
3. avoid destructive migration assumptions,
4. add compatibility notes in changelog/docs,
5. test on partial/legacy documents.

6.5 5. UI/UX contribution expectations

- preserve role-based action clarity,
- keep destructive actions visually explicit and confirmed,
- keep loaders and refresh behavior consistent,
- ensure accessibility/readability in light/dark themes,
- avoid regressions in multilingual rendering.

6.6 6. PR checklist

Before merge:

1. build passes (assembleDebug at minimum),
2. unit tests pass,
3. permission-sensitive flows verified,
4. accounting/reporting side effects reviewed,
5. docs and changelog updated,
6. no unrelated regressions introduced.

6.7 7. Review focus areas

Reviewers should prioritize: - behavioral regressions, - data corruption risks, - permission bypass opportunities, - notification recipient mistakes, - report aggregation correctness.

7 Release Baseline and Changelog Policy

7.1 Why this document exists

Njangui evolves through complex business rules (governance, finances, sanctions, notifications). A clear release baseline is required so teams can: - compare behavior across milestones, - protect schema compatibility, - track risks introduced by new workflows.

7.2 Baseline release snapshot

Date: 2026-04-06
Baseline tag: 1.1.0−baseline

- versionName: 1.1.0
- versionCode: 4
- code name: AJEMI−UDs

7.3 Baseline functional scope

This baseline includes:

1. Authentication and account profile/preferences.
2. Group lifecycle (membership, admins, group rename/currency, leave/delete).
3. Governance lifecycle (staff registration, elections, vote, replacement approval).
4. Tontine lifecycle (ranking preconditions, draw closure, rounds and end state).
5. Caisse lifecycle (individual/shared, accounting operations, transaction history/export).
6. Events and main levees (individual/collective outflow logic and rollback support).
7. Penalties lifecycle (validation, payment, recovery closure).
8. Finance and disciplinary reports with filtered export flows.
9. Background lifecycle sync + notification pull worker.

7.4 Baseline technical posture

- Rich-client domain logic with Firebase persistence.
- WorkManager periodic + immediate scheduling for sync/notifications.
- Rights-aware mutation gates using centralized permission checks.
- Schema tolerance for partially migrated Firestore documents.

7.5 Changelog policy for future milestones

Each release entry must include:

1. release identifiers (versionName, versionCode, date),
2. functional changes visible to users,
3. architectural/data-flow changes,
4. compatibility/migration notes,
5. test and quality impact summary,
6. known risks and mitigations.

7.6 Suggested changelog section template

```
## [x.y.z] – YYYY-MM-DD
### Added
### Changed
### Fixed
### Compatibility
### Quality
```

7.7 Current baseline entry

7.7.1 2026-04-06 - `1.1.0 (4)` - Initial Baseline

Functional: - major group finance and governance workflows delivered, - multilingual UI (fr/en) and theme support, - filtered report exports available.

Technical: - documentation pipeline outputs FR/EN functional and EN technical guides, - GitHub Pages publication from docs/build, - worker-driven background synchronization integrated.

Quality: - baseline unit tests active, - release notes and documentation synchronized.

8 Rule-to-Code Traceability Matrix

Purpose: accelerate audits by linking business rules to implementation anchors.

8.1 1. Governance and rights

Concern	Main components	Outcome
Rights decisions	PermissionsChecker	Grants/denies by member status, mandate, delegated rights
Group and member mutations	TontineGroupPortImpl	Invitations, membership updates, group admin mutations
Staff and elections	HomeActivity, MembersStaffActivity, election models	Staff lifecycle and election persistence

8.2 2. Tontines and ranking

Concern	Main components	Outcome
Tontine CRUD	TontinesPortImpl	Create/update/delete with rights gates
Ranking preconditions	RankingPreConditionsModel, TontinesPortImpl	Strategy, fixed positions, deadlines
Ranking closure	TontinesPortImpl	Auto-complete missing picks and finalize ranking
Round closure	TontinesPortImpl	Validates contributions/outflows and triggers side effects

8.3 3. Caisse accounting

Concern	Main components	Outcome
Caisse operations	CaissesPortImpl	Individual/shared caisse mutations
Accounting invariants	CaisseAccountingUtils	Consistent debit/credit effects
Transaction filtering/export	CaissesActivity	Filtered view and export contract

8.4 4. Events and main levees

Concern	Main components	Outcome
Event lifecycle	EventsPortImpl, EventsActivity	Creation, outflow, rollback, closure
Contribution mode handling	GroupEventModel, EventsPortImpl	Individual vs collective enforcement
Main levee closure	MainLeveeModel, EventsPortImpl	Contribution review and sanctions generation

8.5 5. Penalties and recovery

Concern	Main components	Outcome
Penalty state machine	PenaltiesPortImpl	Validation, payment, recovery closure
Fine payloads	PenaltyFineModel	Amount + text sanctions
Recovery routing	PenaltyRecoveryAllocationModel, PenaltiesPortImpl	Allocation to member/shared/external destinations
Penalty reporting UI	PenaltiesActivity	Filtered disciplinary report export

8.6 6. Reporting

Concern	Main components	Outcome
Finance aggregation	FinancePortImpl	Consolidated financial entries and summaries
Finance export flow	FinanceActivity	Single export from filtered dataset
Caisse transaction export	CaissesActivity	Dialog-based filtered transaction export

8.7 7. Notifications and background sync

Concern	Main components	Outcome
Notification typing and rendering	NotificationTypesAndProcessors	Payload-to-device notification mapping
Background pull and sync	NotificationPullWorker	Periodic lifecycle sync + unread pull
Scheduling	NjanguiApp, BootReceiver	Startup + reboot resilience
Notification data model	NotificationModel	Recipient targeting and delivery tracking

8.8 8. Risk hotspots and guardrails

- Schema compatibility: nullable fields, defensive parsing, migration-safe updates.
- Permission bypass: centralized checks before critical mutations.
- Accounting inconsistency: transaction model + shared utility invariants.
- Notification correctness: recipient filtering, duplicate protection, worker retries.

8.9 9. Suggested audit path for new features

1. Identify impacted business rule.
2. Confirm permission and lifecycle constraints.
3. Verify mutation + notification + reporting coherence.
4. Test with partial legacy-like documents.
5. Update functional and technical docs in same PR.

9 Backend v1.2 Public Release Foundation

9.1 Scope

This chapter documents the v1.2 foundation delivered so far: - migration path from Firebase adapters to a public backend, - modular monolith (hexagonal style, same principles as MBOAGEB), - identity server baseline, - subscription and sponsor model linked to Njangui groups, - public API deployment + admin dashboard deployment.

9.2 Repository layout

The repository is now split as: - src/frontend/mobile/ for the Android app, - src/frontend/dashboard/ for the admin back-office, - src/backend/ for the public backend, - deployment/ for centralized docker/nginx deployment assets, - docs/ as the **single documentation source at repository root**.

9.3 Architecture style

The backend uses a modular monolith with strict layering: - domain - application - infrastructure

Current top-level backend modules: - kernel - common - gateway - identity - subscription - njangui (business bounded contexts scaffolded) - njangui—application (runtime assembly) - tests

9.4 Public domains and hosting

Public domains (same machine, Nginx host-based routing): - API demo: demo.njanguiapp.com - API prod: api.njanguiapp.com - Admin demo: demo—admin.njanguiapp.com - Admin prod: admin.njanguiapp.com

9.5 Identity baseline

Implemented: - account registration/login/refresh token flow, - JWT access + refresh strategy, - stateless security filter, - bootstrap platform admin account, - password reset full flow (expirable link + backend confirmation + landing redirection), - connected device/session listing and revocation, - MFA support: - email OTP challenge (configurable), - TOTP setup/activation with authenticator apps.

Public auth endpoints: - POST /api/v1/public/identity/register - POST /api/v1/public/identity/login - POST /api/v1/public/identity/refresh - GET /api/v1/public/identity/ping - POST /api/v1/public/identity/pas - GET /api/v1/public/identity/password−reset/confirm - POST /api/v1/public/identity/password−reset/comple

Authenticated identity security endpoints: - GET /api/v1/identity/sessions - DELETE /api/v1/identity/sessions/ - DELETE /api/v1/identity/sessions - POST /api/v1/identity/mfa/totp/setup - POST /api/v1/identity/mfa/totp

Login response behavior: - 200 OK with tokens when authentication is fully completed, - 202 Accepted with status metadata when MFA step-up is required (EMAIL_OTP_REQUIRED or TOTP_REQUIRED).

9.6 Gateway and endpoint protection baseline

Implemented gateway-inspired foundations (aligned with MBOAGEB principles): - centralized request locale resolution using request filters (X−Locale then Accept−Language fallback), - thread-local locale holder exposed to all modules through a resolver component, - standardized client context extraction (request id, device/session/client headers, user-agent, resolved client IP), - request-id propagation through response headers, - @GatewayAction metadata + interceptor-based endpoint-level protection checks.

These foundations coexist with Spring Security JWT enforcement for authenticated routes.

9.7 API discoverability (Swagger/OpenAPI)

OpenAPI is now explicitly configured and integrated. - API docs JSON: /v3/api−docs - Swagger UI: /swagger−ui.html

Profile behavior: - dev and demo: enabled - prod: disabled by default

Identity and contact endpoints are now annotated with OpenAPI operation metadata for precise endpoint documentation.

9.8 Subscription model (group-linked)

Implemented capabilities model linked to each Njangui group: - maxMembers - maxTontines - maxNamesPerParticipant - maxCaisses - outflowRedirectionEnabled - autoPenaltiesEnabled

Supported plans: - FREE (default fallback for every group) - PREMIUM−MONTHLY (500 XAF/-month) - PREMIUM−YEARLY (5000 XAF/year)

Current baseline values: - Free: members=25, tontines=25, names/participant=1, caisses=3, redirection disabled, auto penalties disabled. - Premium: limits unlimited (null-based), redirection enabled, auto penalties enabled.

Resolution priority: 1. active sponsor grant for the group/date, 2. active paid group subscription, 3. default free plan.

9.9 Sponsor cards

Sponsor access is implemented through sponsor cards: - card code, - duration in days, - max number of group activations, - usage counter, - optional expiration date, - active/inactive status.

When a sponsor code is activated for a group, usage is incremented and a dated grant is created.

API endpoints: - GET /api/v1/public/subscriptions/plans - GET /api/v1/subscriptions/groups/{groupId}/access
- GET /api/v1/subscriptions/groups/{groupId}/capabilities - PUT /api/v1/subscriptions/groups/{groupId}
- POST /api/v1/subscriptions/sponsor−grants - POST /api/v1/subscriptions/sponsor−codes/activate
- GET /api/v1/subscriptions/sponsor−cards (platform admin) - POST /api/v1/subscriptions/sponsor−cards
(platform admin) - PUT /api/v1/subscriptions/sponsor−cards/{code}/status (platform admin)

9.10 Admin dashboard (React)

Initialized in src/frontend/dashboard/: - React + Vite runtime, - Nginx static serving in Docker, - environment-aware deployment (demo/prod), - live read of public subscription plans endpoint.

9.11 Data storage and migrations

Database: - PostgreSQL

Migrations: - Liquibase from day one, - baseline tables for identity + subscription + sponsor cards + grants.

9.12 Mail infrastructure

Configured external infrastructure: - SMTP: smtp.hostinger.com:465 (SSL/TLS) - IMAP: imap.hostinger.com:993 (SSL/TLS) - POP: pop.hostinger.com:995 (SSL/TLS)

Official contact: - contact@njanguiapp.com

9.13 Tests

Added tests in the same spirit as the MBOAGEB backend service-level validation: - IdentityAuthenticationServiceTe
- SubscriptionManagementServiceTest

Current CI-local verification used during this iteration: - mvn −q −DskipTests compile (backend) - mvn −q test (backend) - npm run build (dashboard)