PORTFOLIO PROJECT STAGE 3 - TECHNICAL DOCUMENTATION (OPEN NAS MANAGER)

September 15, 2025 - September 26, 2025 - Ivane Bagashvili

0. Define User Stories and Mockups (Open NAS Manager)

Purpose

Identify and prioritize MVP features from the user perspective and provide the main screen mockups.

Scope (MVP)

LAN · One TrueNAS server · TrueNAS WebSocket API v2 (JSON-RPC over HTTPS) · API token stored via OS secure storage (Keystore/Keychain) · SQLite for non sensitive data (server info, KPIs, service states, settings).

1) User Stories (MoSCoW)

Must Have

1. Add Server (URL/IP) — As a NAS owner, I add the server URL/IP with optional port and SSL, so the app knows where to connect.

2. Provide API Token (secure storage) — As a NAS owner, I enter a TrueNAS API token stored only in secure storage, so secrets are never exposed in plaintext.

3. Test Connection before Save — As a NAS owner, I test WebSocket v2, TLS/self signed (dev mode), DNS/MagicDNS, and latency before saving, so invalid configurations are blocked.

4. Connect and First Fetch — As a NAS owner, I connect and fetch initial status, so I confirm the server is reachable.

5. Dashboard KPIs — As a NAS owner, I see CPU%, RAM%, and uptime on open, so I can assess health quickly. (CPU/RAM via realtime WS).

6. Service Toggles (SMB/NFS/SSH) — As a NAS owner, I enable/disable SMB, NFS, and SSH in ≤ 3 taps, so routine actions are fast.

7. Pull-to-Refresh — As a NAS owner, I swipe down to refresh KPIs and service states, so data stays current.

8. Success/Error Feedback — As a NAS owner, I get clear success and error messages (401/403/5xx/TLS/DNS), so I know the result of an action.

9. Error Banners (auth/TLS/network) — As a NAS owner, I see dedicated banners for invalid token, TLS/self-signed required, or network loss, so I can react.

10. SQLite Offline Cache — As a NAS owner, I get last known KPIs and service states from SQLite at launch, so the app remains useful offline.

11. Edit Server — As a NAS owner, I edit URL/IP/port/SSL and re-test, so I can fix the configuration later.

12. Logout & Wipe — As a NAS owner, I log out to wipe the token and session data, so the device remains safe.

Should Have

13. Connection Badge (LAN / Tailscale) — As a NAS owner, I see LAN vs Tailscale in the header, so I know the connectivity context.

14. Diagnostics Panel — As a NAS owner, I open diagnostics (DNS/MagicDNS, 100.x, TLS), so I can troubleshoot.

15. Auto-Refresh — As a NAS owner, I enable periodic refresh (e.g., every 30–120 s), so KPIs stay up to date (in addition to realtime).

16. Theme (System/Light/Dark) — As a NAS owner, I choose a theme, so readability matches my preference.

17. Idle Timeout — As a NAS owner, the app locks after inactivity (configurable), so access is limited on idle.

18. Undo Last Toggle — As a NAS owner, I undo the last service toggle from a snackbar, so I can correct a mistake.

Could Have

19. Storage View (read-only) — As a NAS owner, I see pools/datasets (capacity/health) read-only, so I understand storage at a glance.

20. Apps List (read-only) — As a NAS owner, I list installed apps and simple resource usage, so I know what is running.

21. KPI Trend (7 days) — As a NAS owner, I see short KPI history stored in SQLite, so I can spot anomalies.

22. In-App Feedback / Export Diagnostics — As a NAS owner, I send feedback or export diagnostics, so support is simpler.

23. Quick Actions (launcher) — As a NAS owner, I long-press the app icon for quick actions (refresh), so I save time.

Won't Have (for this MVP)

24. Multi-server management

25. ZFS/dataset writes, backups/snapshots orchestration

26. Voice assistant flows

Prioritization

Must Have: 1–12 (directly support the SMART goals: time-to-status ≤ 2s, secure login, service control ≤ 3 taps).

Should Have: 13–18 (improve usability and resilience without blocking MVP).

Could Have: 19–23 (optional enhancements, if time permits).

Won't Have: 24–26 (explicitly excluded to protect scope).

Acceptance Criteria (Must)

Connection & Security (US 1–4, 11–12)

• After a successful Test, persist host/port/useTls/name (label)/ acceptSelfSigned (development-only)/createdAt/lastConnected in SQLite.

• Token stored only in secure storage; never in SQLite or logs.

• Edit Server requires re test for sensitive changes (host/port/SSL/ acceptSelfSigned (development-only)).

• Logout wipes: disconnect WS, clear current server id, delete API token for the current server from secure storage.

Test Connection (US 3)

• Show checks: WebSocket v2 ✓ · TLS ✓ (self-signed allowed in development builds only) · DNS/MagicDNS ✓ · Latency ms · 'Signed in as <account>' when available.
(Production builds: self-signed NOT allowed; valid TLS required).

• If a check fails → Save disabled + clear guidance.

Performance & Display (US 5, 7, 10)

• Skeletons render < 150 ms after open.

• Time-to-status ≤ 2 s on LAN.

• Offline → show last snapshot from SQLite ≤ 300 ms, then update with network data.

Toggles & Feedback (US 6, 8, 9)

• SMB/NFS/SSH in ≤ 3 taps; success snackbar; error explains 401/403/5xx/TLS/DNS/network.

• Dedicated banners for invalid token / TLS required / network lost.

• UI state consistent with API after refresh.

Data Model (SQLite — non-sensitive only)

• Servers: id PK; name(label) TEXT; host TEXT; port INTEGER; ssl(useTls) INTEGER; acceptSelfSigned INTEGER [development-only]; createdAt INT(ms); lastConnected INT(ms); UNIQUE(host, port).

• SystemInfo (current snapshot): id PK; serverId FK→Servers.id; hostname TEXT?; version TEXT?; uptimeSeconds INT?; cpuUsagePct REAL?; memoryUsagePct REAL?; updatedAt INT(ms); UNIQUE(serverId).

• ServiceState: id PK; serverId FK→Servers.id; serviceName TEXT (SMB|NFS|SSH…); isRunning INTEGER; autoStart INTEGER?; updatedAt INT(ms); UNIQUE(serverId, serviceName).

• Settings: key TEXT PK; value TEXT (e.g., theme, autoRefresh, idleTimeout).

Notes:

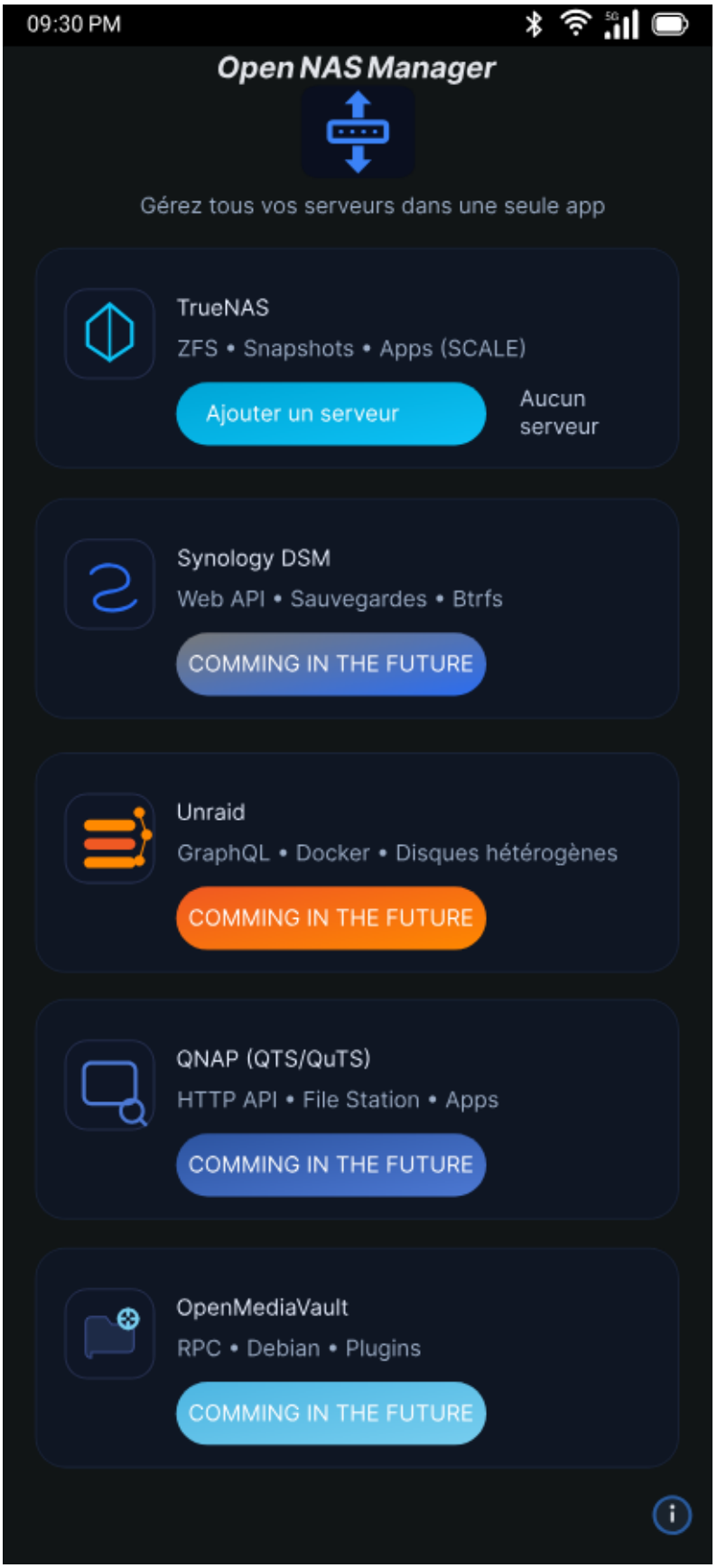
- All timestamps are epoch milliseconds.
- Booleans stored as INTEGER 0/1.
- Children tables use ON DELETE CASCADE to clean up when a server is removed.
- API token is never stored in SQLite (Secure Storage only).

Future

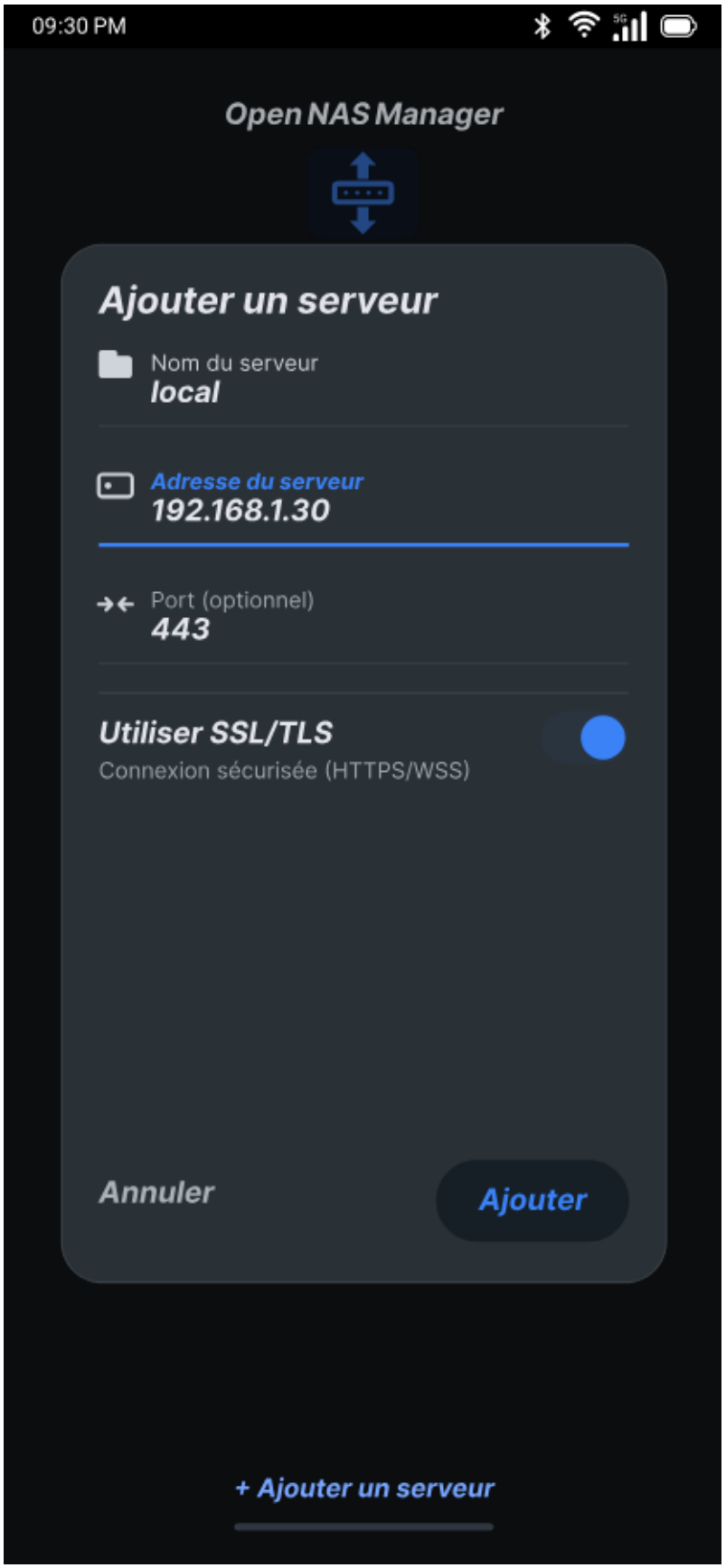
- KpiSnapshot: id PK; serverId FK; cpuPct REAL?; ramPct REAL?; uptimeSec INT?; takenAt INT(ms); INDEX(serverId, takenAt DESC)
- Pools: id PK; serverId FK; name TEXT; health TEXT; capacityPct REAL?; usedBytes INT?; availableBytes INT?; updatedAt INT(ms); UNIQUE(serverId, name)
- Apps: id PK; serverId FK; name TEXT; status TEXT; cpuPct REAL?; ramMiB REAL?; hasUpdate INTEGER?; updatedAt INT(ms); UNIQUE(serverId, name)
- Diagnostics: id PK; serverId FK; lastLatencyMs INT?; lastSeenAt INT?; isTailscale INTEGER?; magicDnsActive INTEGER?; truenasAccount TEXT?; updatedAt INT(ms); UNIQUE(serverId)
- Certificates: id PK; serverId FK; certFingerprintSha256 TEXT; pinned INTEGER DEFAULT 0; updatedAt INT(ms); UNIQUE(serverId)

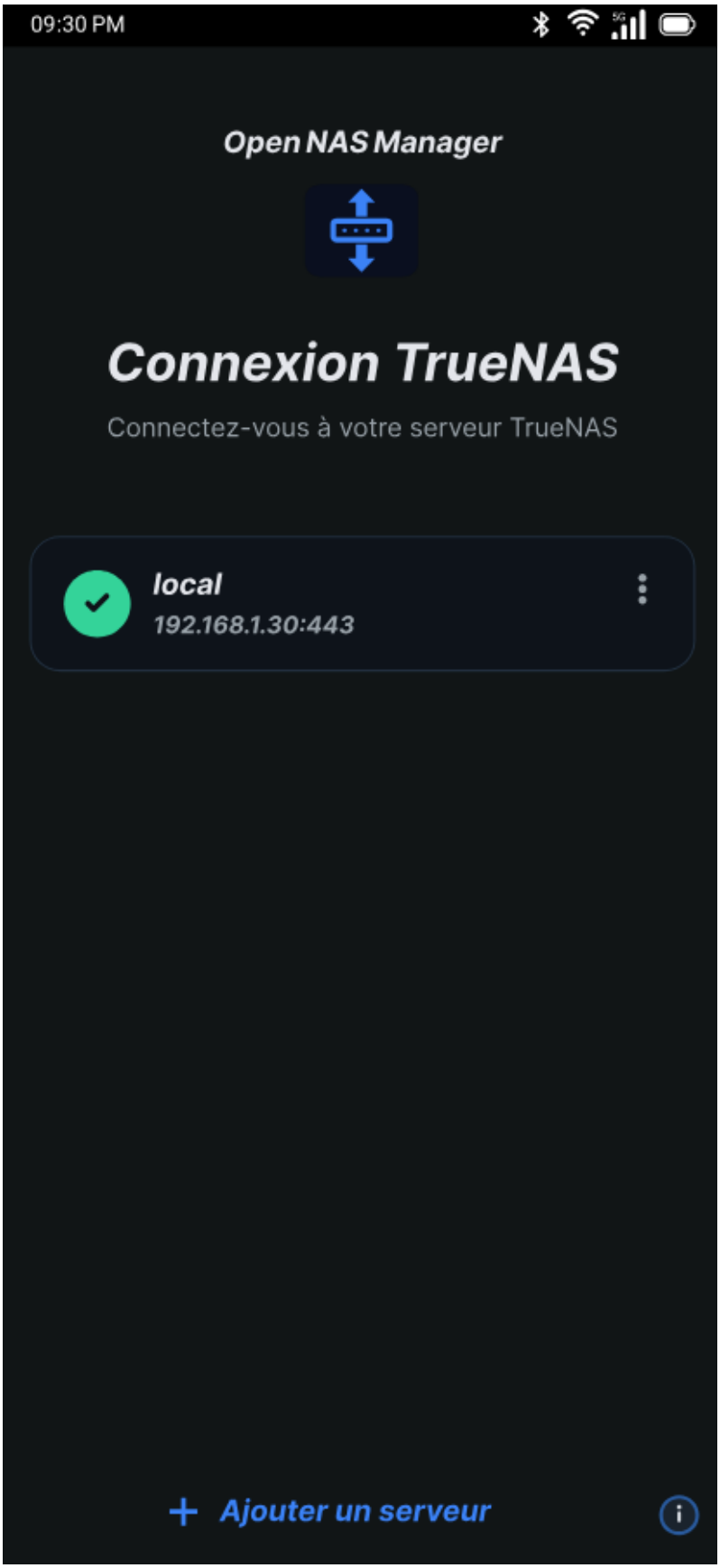
Mockups

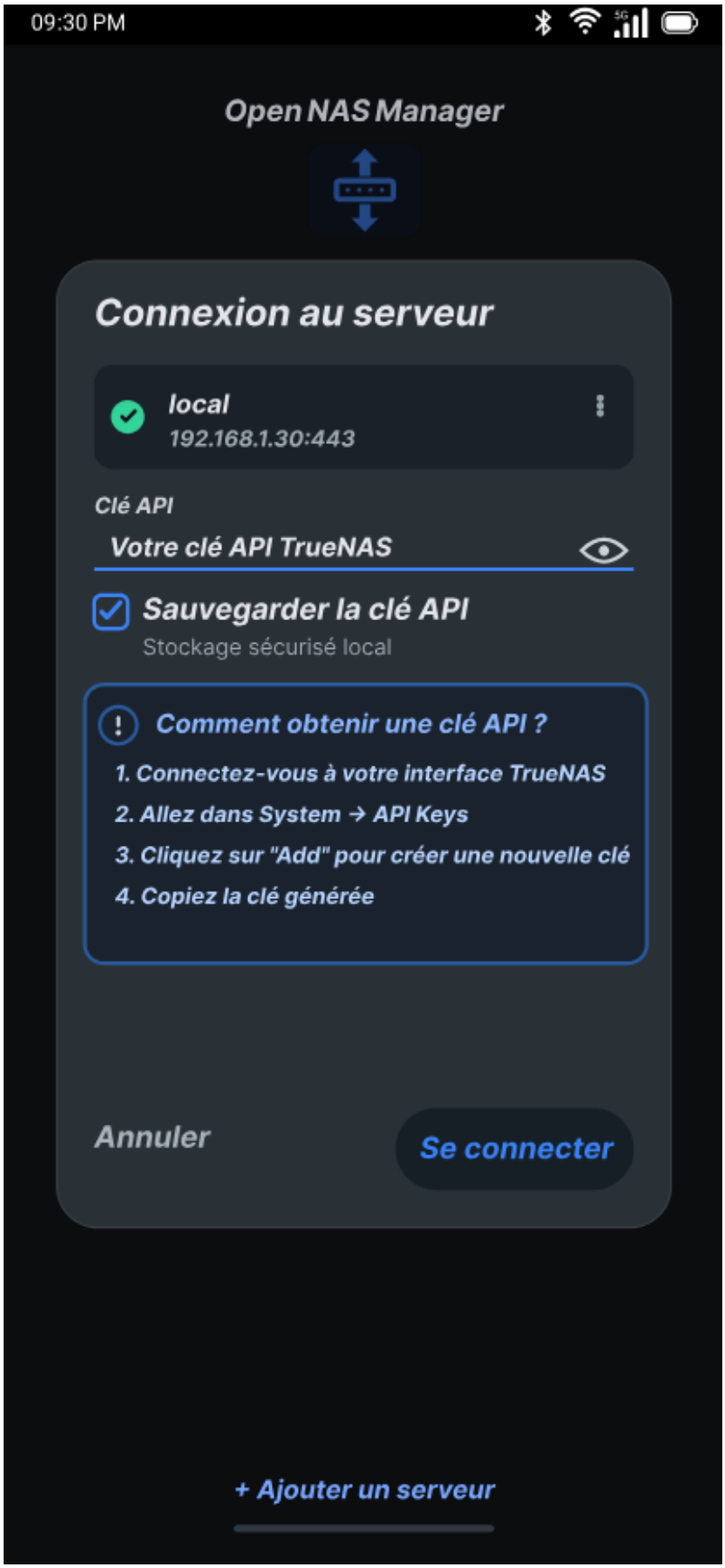
Main screens (Figma or PDF): Add Server / Connect (URL/IP, Port, SSL, Token, Test, Save); Dashboard (KPIs + Services SMB/NFS/SSH, pull-to-refresh, banners); Settings (Edit server, Auto-refresh, Theme, Idle timeout, Logout); Optional read-only: Storage / Apps.

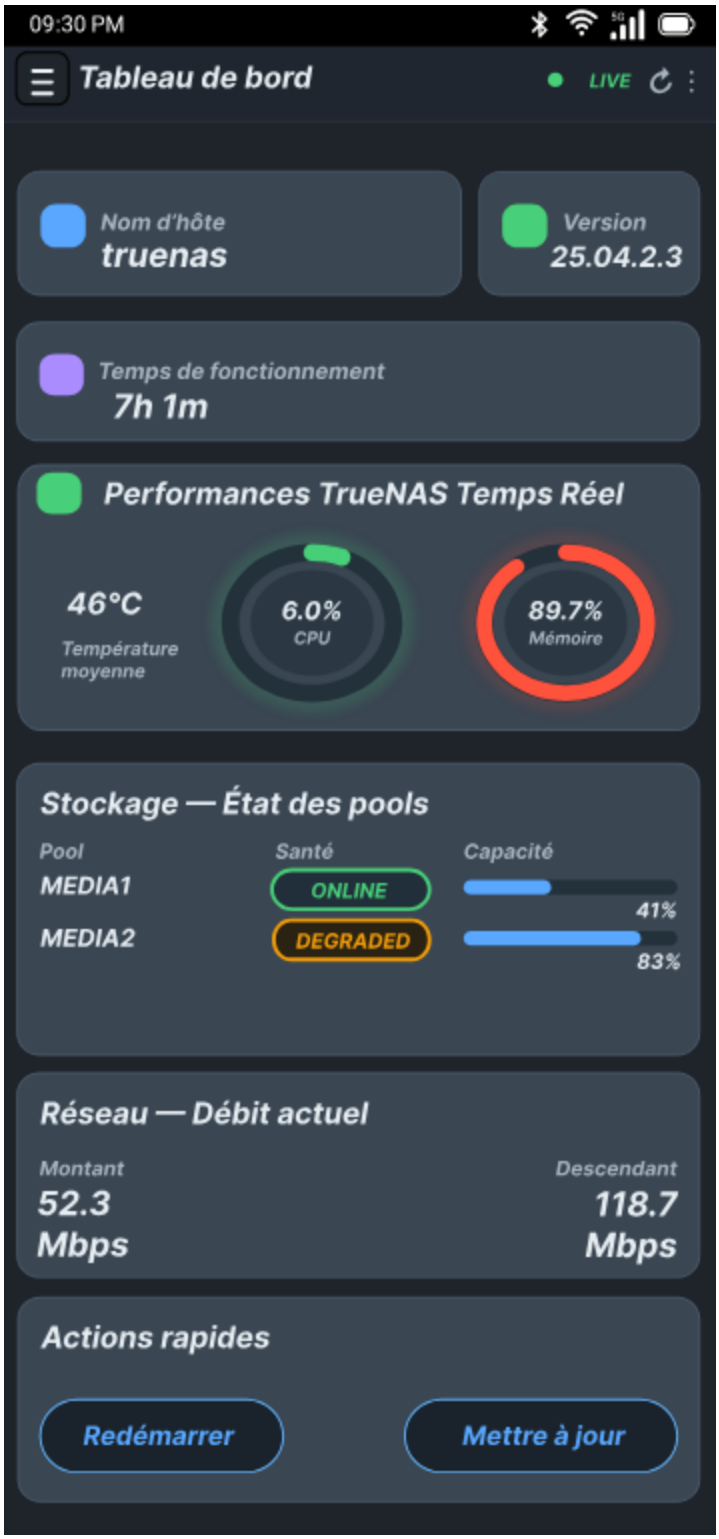

09:30 PM
Open NAS Manager
Gérez tous vos serveurs dans une seule app
TrueNAS
ZFS • Snapshots • Apps (SCALE)
Ajouter un serveur
Aucun serveur
Synology DSM
Web API • Sauvegardes • Btrfs
COMMING IN THE FUTURE
Unraid
GraphQL • Docker • Disques hétérogènes
COMMING IN THE FUTURE
QNAP (QTS/QuTS)
HTTP API • File Station • Apps
COMMING IN THE FUTURE
OpenMediaVault
RPC • Debian • Plugins
COMMING IN THE FUTURE


09:30 PM
Open NAS Manager
Connexion TrueNAS
Connectez-vous à votre serveur TrueNAS
Ajouter un serveur
Aucun serveur configuré
Ajoutez votre premier serveur TrueNAS


09:30 PM
Open NAS Manager
Ajouter un serveur
Nom du serveur
local
Adresse du serveur
192.168.1.30
Port (optionnel)
443
Utiliser SSL/TLS
Connexion sécurisée (HTTPS/WSS)
Annuler
Ajouter
+ Ajouter un serveur

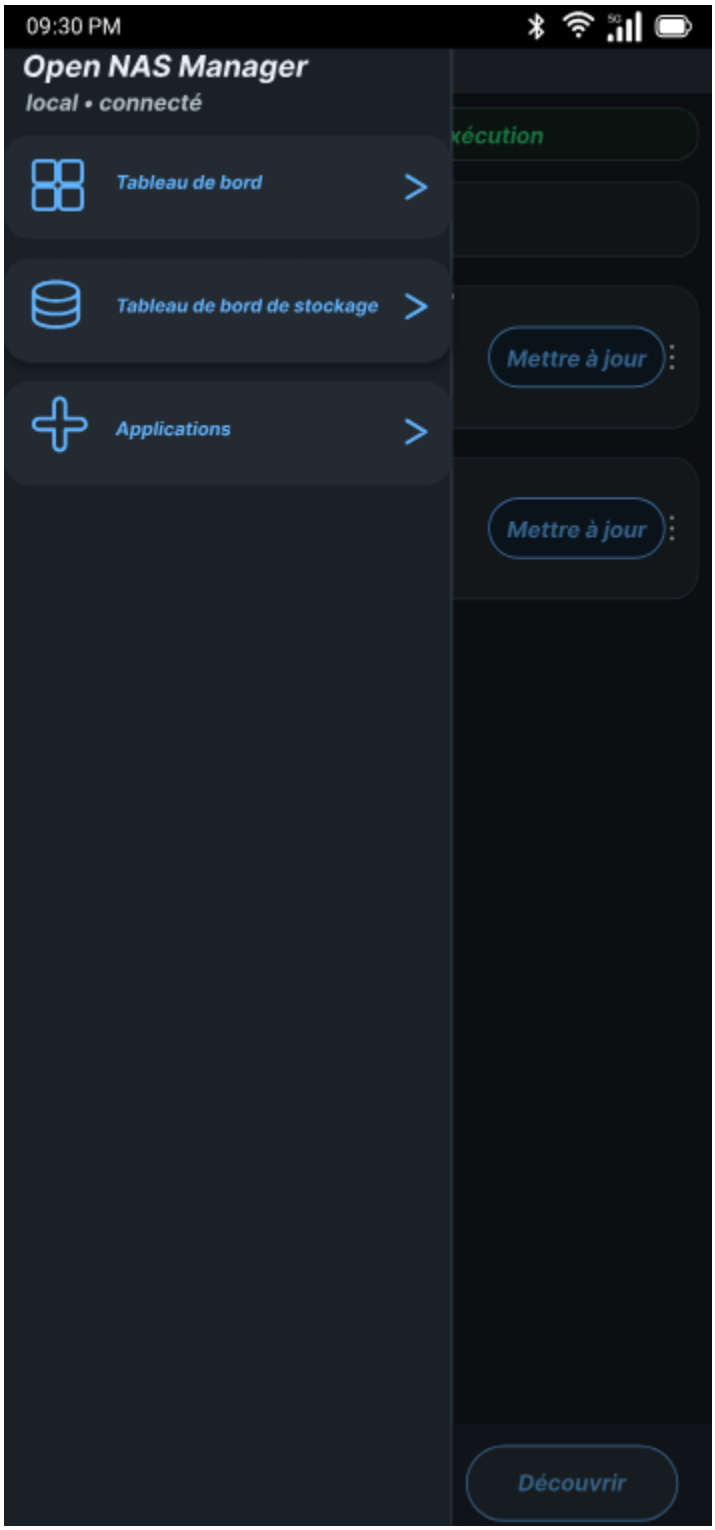

09:30 PM
Open NAS Manager
Connexion TrueNAS
Connectez-vous à votre serveur TrueNAS
local
192.168.1.30:443
Ajouter un serveur

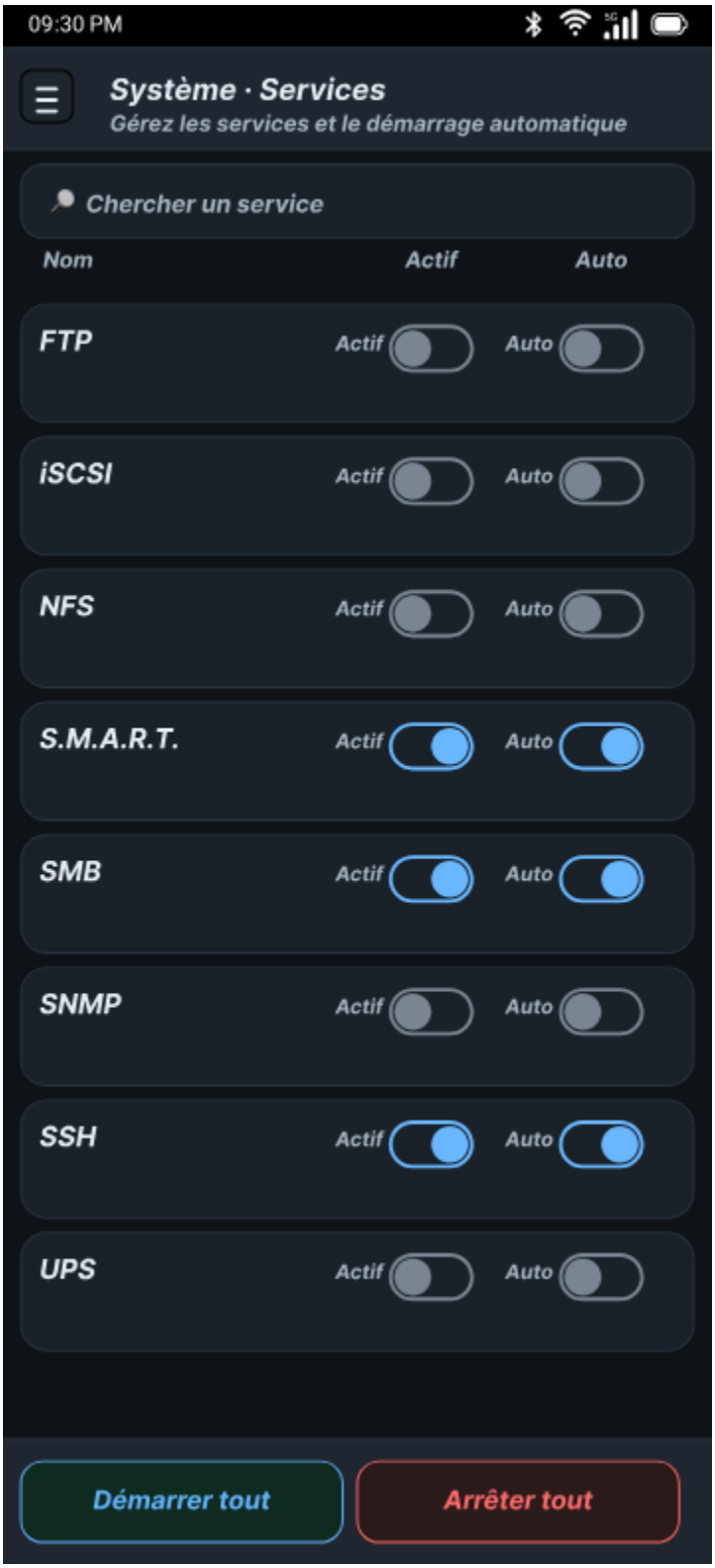

09:30 PM
Open NAS Manager
Connexion au serveur
local
192.168.1.30:443
Clé API
Votre clé API TrueNAS
Sauvegarder la clé API
Stockage sécurisé local
Comment obtenir une clé API ?
1. Connectez-vous à votre interface TrueNAS
2. Allez dans System → API Keys
3. Cliquez sur "Add" pour créer une nouvelle clé
4. Copiez la clé générée
Annuler
Se connecter
+ Ajouter un serveur


09:30 PM
Tableau de bord
LIVE
Nom d'hôte
truenas
Version
25.04.2.3
Temps de fonctionnement
7h 1m
Performances TrueNAS Temps Réel
46°C
Température moyenne
6.0%
CPU
89.7%
Mémoire
Stockage — État des pools
Pool
Santé
Capacité
MEDIA1
ONLINE
41%
MEDIA2
DEGRADED
83%
Réseau — Débit actuel
Montant
52.3 Mbps
Descendant
118.7 Mbps
Actions rapides
Redémarrer
Mettre à jour

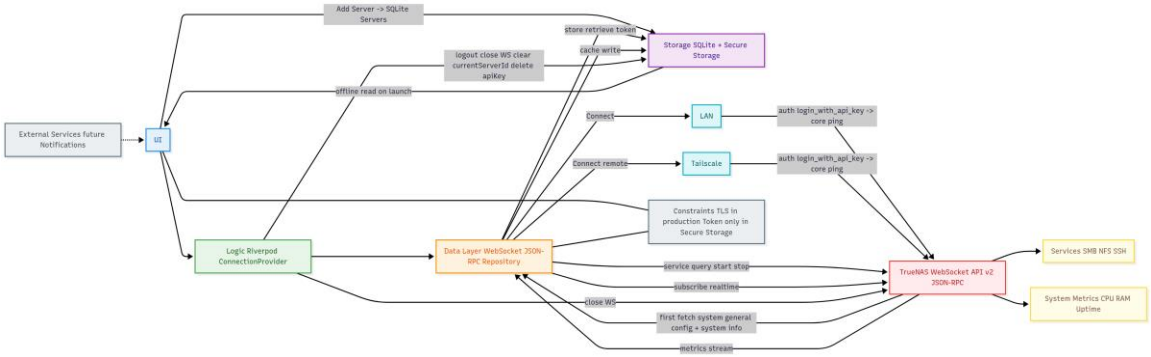

09:30 PM
Tableau de bord du stockage
Importer
Créer volume
MEDIA1
Exporter/Déconn.
Étendre
Utilisation
Gérer datasets
78.5 %
Capacité utile: 899.25 GiB
Utilisé: 706.0 GiB
Disponible: 193.2 GiB
Santé ZFS
Nettoyage
ONLINE
État: En ligne • Erreurs: 0 • Auto TRIM: On
Santé du disque
Gérer disques
Alertes temp.: 0 • Moyenne: —
MEDIA2
Exporter/Déconn.
Étendre
Utilisation
Gérer datasets
77.5 %
Capacité utile: 899.25 GiB
Utilisé: 696.1 GiB
Disponible: 203.14 GiB
Santé ZFS
Nettoyage
DEGRADED
État: En ligne • Erreurs: 0 • Auto TRIM: Off
Santé du disque
Gérer disques
Alertes temp.: 0 • Moyenne: —


09:30 PM
Applications installées
Service d'applications en cours d'exécution
Chercher
Immich
Actif
Mettre à jour
CPU 0% • RAM 412.0 MiB
Tailscale
Actif
Mettre à jour
CPU 0% • RAM 68.3 MiB
Redémarrer
Tout mettre à jour
Découvrir


09:30 PM
Open NAS Manager
local • connecté
Tableau de bord
Tableau de bord de stockage
Applications
xécution
Mettre à jour
Mettre à jour
Découvrir


09:30 PM
Système · Services
Gérez les services et le démarrage automatique
Chercher un service
Nom
Actif
Auto
FTP
Actif
Auto
iSCSI
Actif
Auto
NFS
Actif
Auto
S.M.A.R.T.
Actif
Auto
SMB
Actif
Auto
SNMP
Actif
Auto
SSH
Actif
Auto
UPS
Actif
Auto
Démarrer tout
Arrêter tout

1. Design System Architecture



The Flutter app separates Presentation, Logic, and Data layers. The API token is stored only in OS Secure Storage (Keystore/Keychain) and never in SQLite. Non-sensitive data (server info/label, KPI cache, service states, settings) is stored in SQLite. The Data layer communicates with TrueNAS WebSocket API v2 (JSON-RPC over HTTPS) via LAN by default, or Tailscale for remote access. Initial data (system.general.config + system.info) is fetched on connect, then KPIs (CPU/RAM) are updated in real time via a WebSocket subscription. A lightweight offline-first cache displays the last snapshot on launch, then the network refresh overwrites it; the API remains the source of truth. Optional integrations (e.g., notifications) are planned for later.

- Add Server: UI → SQLite.Servers
- Connect: Data Layer → WS auth.login_with_api_key → core.ping; token → Secure Storage
- First Fetch: API → system.general.config + system.info → Data Layer → SQLite (cache) → UI
- Realtime: Data Layer → subscribe reporting.realtime; API → KPI metrics stream → Data Layer → UI
- Services: Data Layer → service.query; toggle → service.start/stop → refresh UI + cache
- Offline-first: on launch UI ← last SQLite snapshot, then network overwrites
- Logout: close WebSocket → clear currentServerId → delete API key from Secure Storage

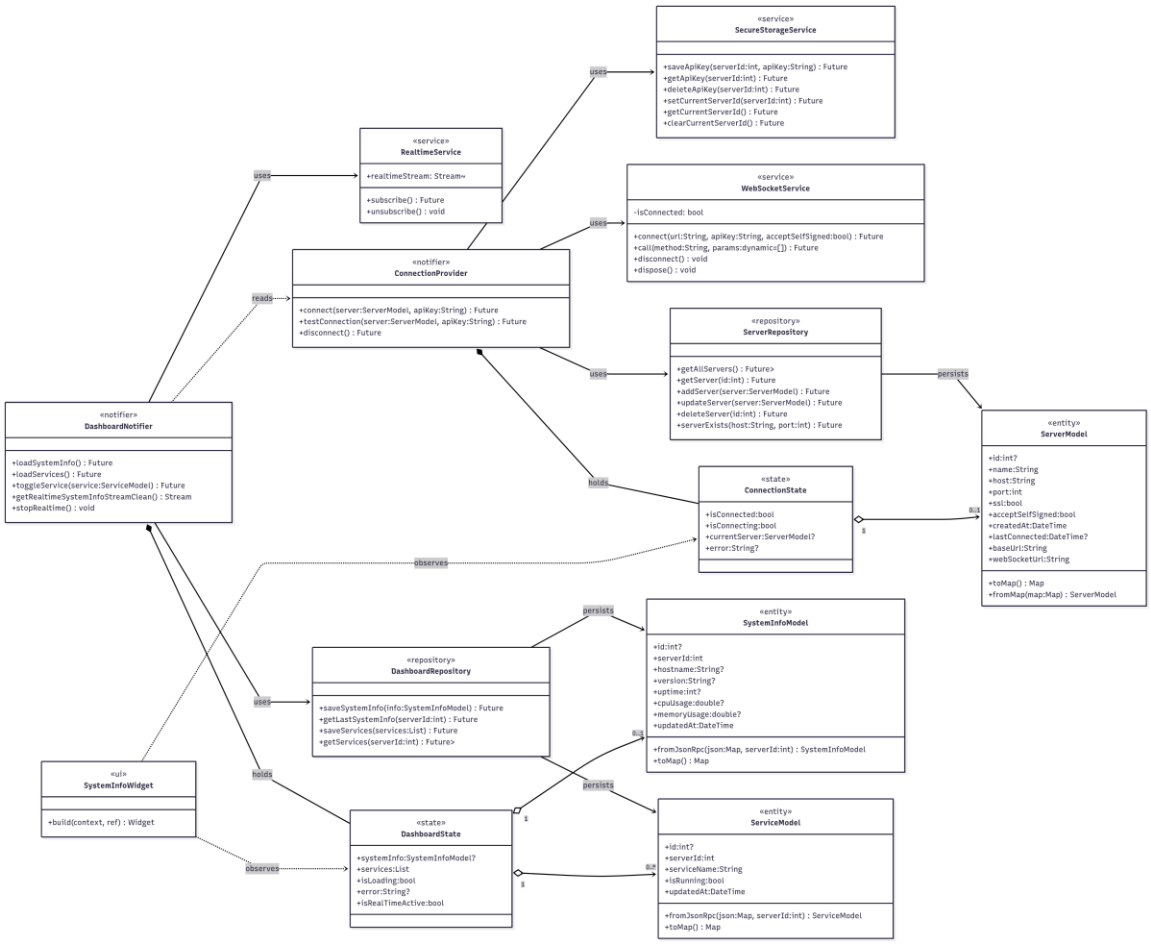
2. Define Components, Classes, and Database Design

Front-end Design Reference

Outline of Main UI Components and Their Interactions
The front-end structure and user interface flows are documented in the following Figma design:

[View Figma Prototype]

Class Diagram



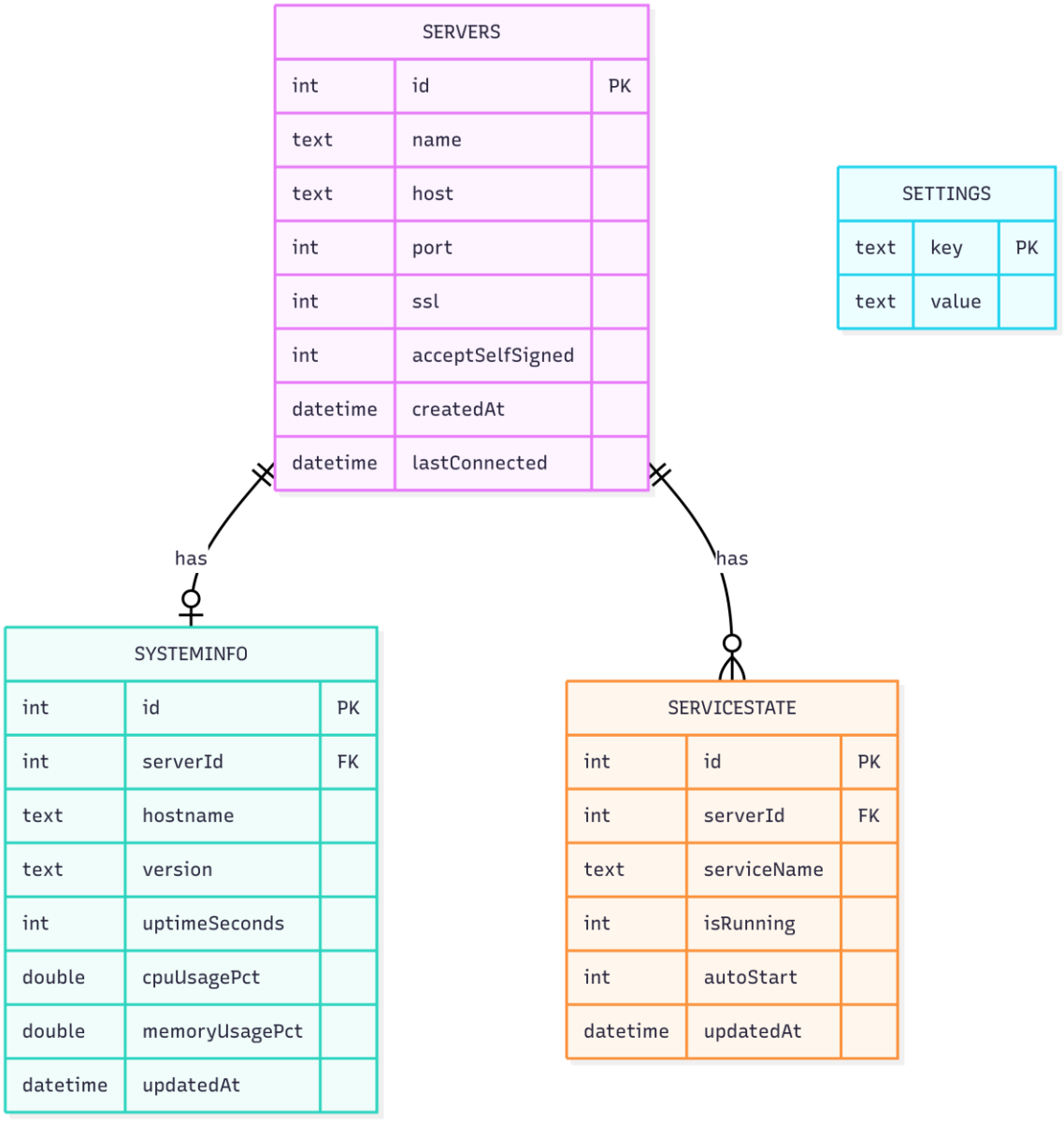
The Class Diagram above illustrates the static structure of the Open NAS Manager management application.

It defines the main entities (ServerModel, SystemInfoModel, ServiceModel) and their supporting components, detailing their attributes, methods, and the relationships that govern their interaction within the system's architecture.Key components:

- **ServerModel**: Represents a TrueNAS server configuration with connection details (host, port, SSL, development mode) and metadata (name, timestamps).
- **SystemInfoModel**: Represents current system metrics (CPU, memory, uptime, hostname, version) fetched from TrueNAS and cached locally.
- **ServiceModel**: Represents individual services (SMB, NFS, SSH) with their running state and metadata.
- **SecureStorageService**: Manages API token storage per server using OS secure storage (Keystore/Keychain), ensuring secrets are never exposed in plaintext.
- **WebSocketService**: Handles JSON-RPC communication over HTTPS with TrueNAS WebSocket API v2, including authentication and method calls.
- **RealtimeService**: Manages real-time data subscriptions for live CPU and memory metrics streaming.
- **Repository classes (ServerRepository, DashboardRepository)**: Encapsulate data persistence operations, managing SQLite storage for non-sensitive data while maintaining separation from secure token storage.
- **State Management (ConnectionProvider, DashboardNotifier)**: Orchestrate business logic, manage application state, and coordinate between UI components and data services.

The diagram clearly defines ownership relationships (notifiers hold state objects), usage dependencies (providers use repositories and services), and data flow patterns (UI observes state via Riverpod, repositories persist entities), ensuring proper separation of concerns and maintainable architecture.

Database Diagram



Database Schema (ERD)

SERVERS table

- id (Primary Key)
- name
- host
- port
- ssl
- acceptSelfSigned
- createdAt
- lastConnected

SYSTEMINFO table

- id (Primary Key)
- serverId (Foreign Key → SERVERS.id)
- hostname
- version
- uptimeSeconds
- cpuUsagePct
- memoryUsagePct
- updatedAt

SERVICESTATE table

- id (Primary Key)
- serverId (Foreign Key → SERVERS.id)
- serviceName
- isRunning
- autoStart
- updatedAt

SETTINGS table

- key (Primary Key)
- value

Relationships

One-to-one between SERVERS and SYSTEMINFO: Each server has one current system info snapshot.

One-to-many between SERVERS and SERVICESTATE: A server can have multiple service states (SMB, NFS, SSH).

Description of Front-End Components

Dashboard page

Role: Displays system KPIs (CPU, memory, uptime) and service states in real-time.

Input data: None, displays cached data from SQLite on launch.

Interaction with the back-end:

• Observes DashboardState via Riverpod

• Calls DashboardNotifier.loadSystemInfo() and loadServices() for initial fetch

• Subscribes to RealtimeService for live CPU/memory updates

Response to replies:

• Shows loading states during fetch

• Displays error banners for connection/auth issues

• Updates KPIs in real-time via WebSocket stream

Server connection page

Role: Allows adding and connecting to TrueNAS servers.

Input data: server name, host, port, SSL settings, API token.

Interaction with the back-end:

• Calls ConnectionProvider.testConnection() before save

• Calls ConnectionProvider.connect() for authentication

• Stores server config in SQLite, API token in Secure Storage

Response to replies:

• Disables save button if test fails

• Shows connection status and error messages

• Redirects to dashboard on successful connection

Services management page

Role: Lists and controls system services (SMB, NFS, SSH).

Input data: Service toggle actions.

Interaction with the back-end:

• Calls DashboardNotifier.toggleService() for start/stop operations

• Observes service states from DashboardState

Response to replies:

• Shows success snackbars for successful toggles

• Displays error messages for failed operations

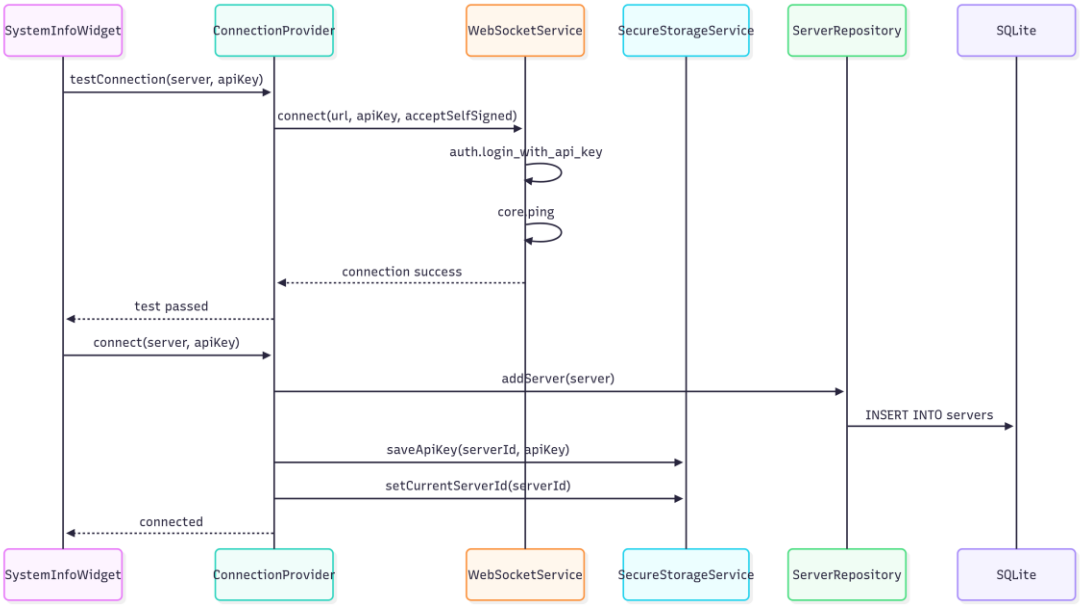
• Updates service status in real-time

3. Create High-Level Sequence Diagrams

This section presents a series of sequence diagrams that illustrate the flow of interactions between the components of the TrueNAS mobile management application in response to key user actions. These diagrams are designed to show how the system's three main layers - Presentation (UI), Business Logic (State Management), and Persistence (Repositories + Storage) - collaborate to fulfill specific user requests. Three diagrams will be presented next:

- Add Server & Connect
- Dashboard Load & Realtime Updates
- Service Toggle

1. Add Server & Connect



The user fills out the server connection form on the front-end by entering server details (name, host, port, SSL settings) and API token. The front-end calls ConnectionProvider.testConnection() to validate the configuration before saving.

The ConnectionProvider receives the request and forwards it to WebSocketService, which establishes a connection to the TrueNAS server using the provided credentials. The WebSocketService performs authentication via auth.login_with_api_key and validates the connection with core.ping.

Once the test connection succeeds, the front-end calls ConnectionProvider.connect() to establish the permanent connection. The ConnectionProvider interacts with ServerRepository to save the server configuration in SQLite, and with SecureStorageService to store the API token securely.

The ConnectionProvider returns the response to the front-end which displays either an error message or redirects to the dashboard on successful connection.

2. Dashboard Load & Realtime Updates

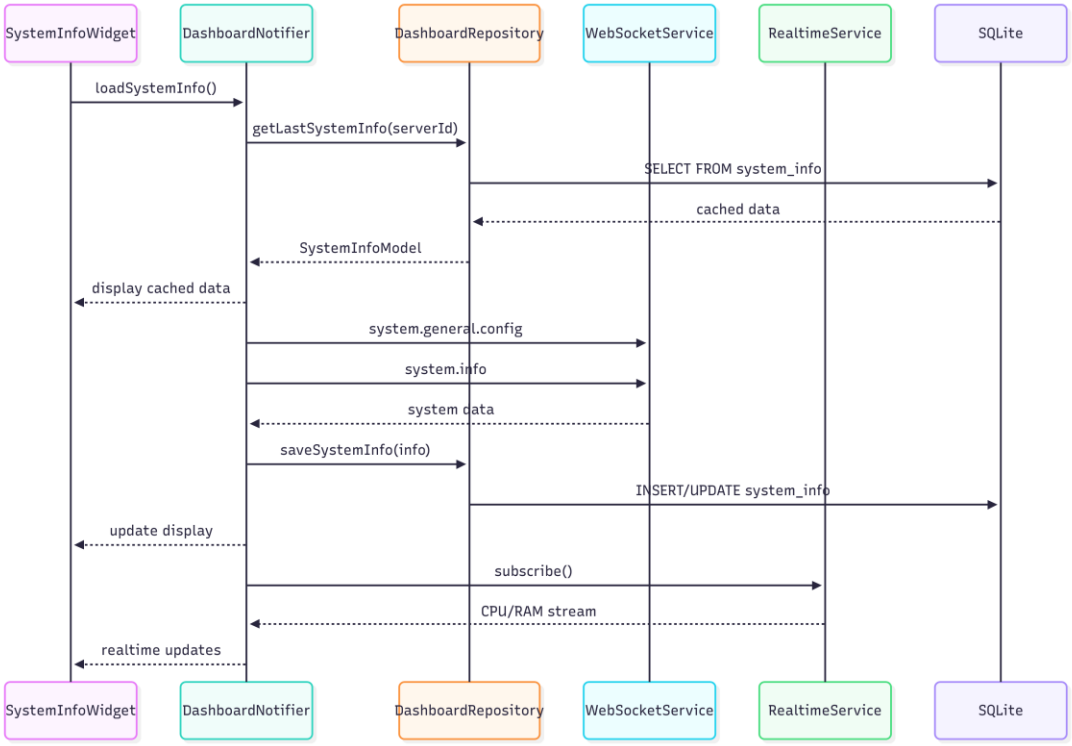
The user opens the dashboard screen which displays system KPIs (CPU, memory, uptime) and service states. The front-end calls DashboardNotifier.loadSystemInfo() to fetch current system information.

The DashboardNotifier first retrieves cached data from DashboardRepository, which queries SQLite for the last known system info snapshot. This provides immediate offline-first display while network data is being fetched.

The DashboardNotifier then establishes network connections to fetch fresh data. It calls WebSocketService to retrieve system.general.config and system.info from the TrueNAS server. The received data is processed and saved to SQLite via DashboardRepository for future offline access.

Simultaneously, the DashboardNotifier subscribes to RealtimeService for live CPU and memory metrics. The RealtimeService maintains a WebSocket subscription that streams real-time performance data.

The DashboardNotifier returns the cached data immediately to the front-end for instant display, then updates the UI with fresh network data, and finally provides continuous real-time updates via the WebSocket stream. The front-end displays loading states during initial fetch and shows error banners for connection issues.



3. Service Toggle

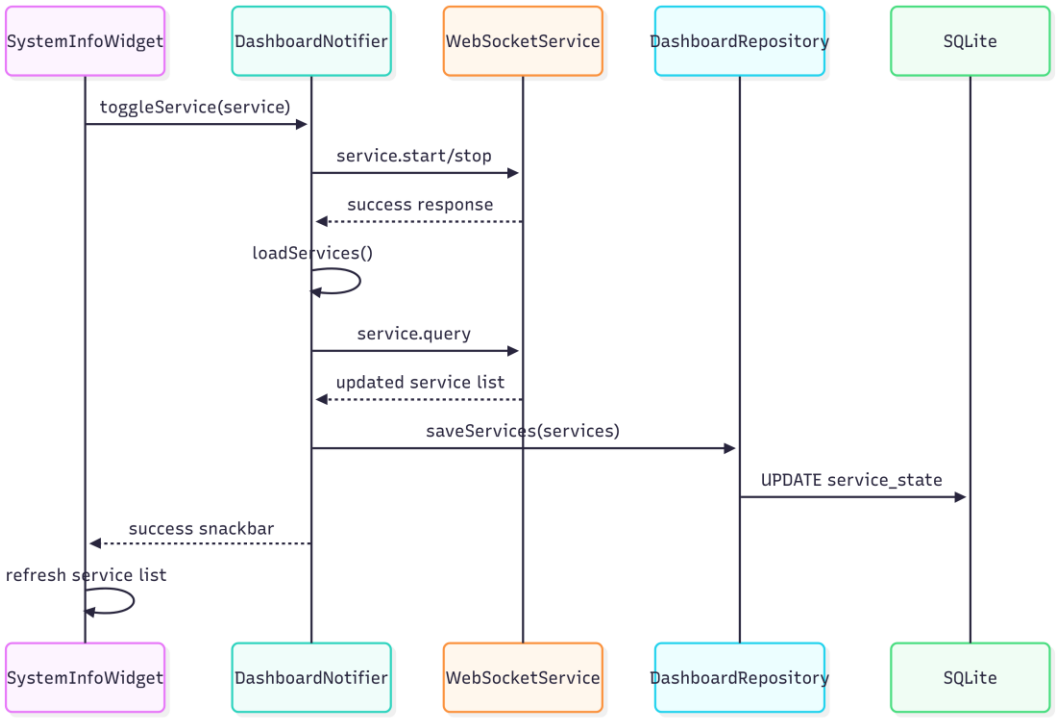
The user interacts with a service toggle (SMB, NFS, or SSH) on the services management screen to start or stop a specific service. The front-end calls DashboardNotifier.toggleService() with the selected service.

The DashboardNotifier receives the request and determines the appropriate action (start or stop) based on the current service state. It calls WebSocketService to execute the service.start or service.stop command on the TrueNAS server.

The WebSocketService communicates with the TrueNAS server via JSON-RPC and returns a success or failure response. Upon successful execution, the DashboardNotifier triggers a refresh by calling loadServices() to fetch the updated service states.

The DashboardNotifier calls WebSocketService.service.query to retrieve the current status of all services, then processes the response and saves the updated service states to SQLite via DashboardRepository for offline caching.

The DashboardNotifier returns a success response to the front-end which displays a confirmation snackbar and refreshes the service list to show the updated status. If the operation fails, an error message is displayed to inform the user of the issue.



4. Document External and Internal APIs

External and Internal APIs

In this section, we will see how the system interacts with external APIs and how it defines its own APIs.

External APIs

The application uses the TrueNAS WebSocket API v2 (JSON-RPC over HTTPS) to communicate with TrueNAS servers. This external API provides real-time system monitoring, service management, and server configuration capabilities. The WebSocket protocol enables live data streaming for CPU and memory metrics, while JSON-RPC offers structured method calls for system operations. HTTPS ensures secure communication with proper TLS validation in production environments.

Internal APIs

For each internal API, we provide Method, Path, Purpose, expected Parameters (JSON when applicable), Output settings, and example Success / Failure responses with HTTP status codes.

Server Management - Connect

Method: POST

Path: ConnectionProvider.connect()

Purpose: Connect to a TrueNAS server

Expected parameters: JSON format

```
{
  "server": {
    "name": "MyServer",
    "host": "192.168.1.10",
    "port": 443,
    "ssl": true,
    "acceptSelfSigned": false
  },
  "apiKey": "API_KEY_EXAMPLE"
}
```

Output settings: JSON format

Success

HTTP Status code: 200 OK

Response body:

```
{
  "message": "Connection established successfully",
  "serverState": "connected",
  "storage": "saved to SQLite",
  "token": "stored in Secure Storage",
  "websocket": "active"
}
```

Failures

HTTP Status code: 400 Bad Request

Response body:

```
{ "message": "Invalid server configuration (host/port format)" }
```

HTTP Status code: 401 Unauthorized

Response body:

```
{ "message": "Authentication failure (invalid API token)" }
```

HTTP Status code: 403 Forbidden

Response body:

```
{ "message": "Server access denied" }
```

HTTP Status code: 404 Not Found

Response body:

```
{ "message": "Server not reachable" }
```

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Network connectivity issues, TLS certificate
validation errors" }
```

Server Management - Test Connection

Method: POST

Path: ConnectionProvider.testConnection()

Purpose: Validate server configuration before saving

Expected parameters: JSON format

```
{
  "server": {
    "name": "MyServer",
    "host": "192.168.1.10",
    "port": 443,
    "ssl": true
  },
  "apiKey": "API_KEY_EXAMPLE"
}
```

Output settings: JSON format

Success

HTTP Status code: 200 OK

Response body:

```
{
  "message": "Test connection successful",
  "serverReachable": true,
  "authentication": "verified",
  "websocket": "established"
}
```

Failures

HTTP Status code: 400 Bad Request

Response body:

```
{ "message": "Invalid host/port format" }
```

HTTP Status code: 401 Unauthorized

Response body:

```
{ "message": "Authentication error" }
```

HTTP Status code: 403 Forbidden

Response body:

```
{ "message": "SSL/TLS issues" }
```

HTTP Status code: 404 Not Found

Response body:

```
{ "message": "DNS resolution failure" }
```

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Network connectivity issues" }
```

Dashboard Data - Load System Info

Method: GET

Path: DashboardNotifier.loadSystemInfo()

Purpose: Fetch system information and KPIs

Expected parameters: No input expected

Output settings: JSON format

Success

HTTP Status code: 200 OK

Response body:

```
{
  "hostname": "truenas.local",
  "version": "TrueNAS-SCALE-23.10",
  "uptimeSeconds": 86400,
  "cpuUsagePct": 12.5,
  "memoryUsagePct": 73.2,
  "updatedAt": "2025-09-26T12:00:00Z"
}
```

Failures

HTTP Status code: 401 Unauthorized

Response body:

```
{ "message": "Authentication problems" }
```

HTTP Status code: 404 Not Found

Response body:

```
{ "message": "Server unreachable" }
```

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Network connectivity issues, API method errors" }
```

Dashboard Data - Load Services

Method: GET

Path: DashboardNotifier.loadServices()

Purpose: Retrieve service states (SMB, NFS, SSH)

Expected parameters: No input expected

Output settings: JSON format

Success

HTTP Status code: 200 OK

Response body:

```
[
  { "serviceName": "SMB", "isRunning": true,  "autoStart": true,
"updatedAt": "2025-09-26T12:00:00Z" },
  { "serviceName": "NFS", "isRunning": false, "autoStart": false,
"updatedAt": "2025-09-26T12:00:00Z" },
  { "serviceName": "SSH", "isRunning": true,  "autoStart": true,
"updatedAt": "2025-09-26T12:00:00Z" }
]
```

Failures

HTTP Status code: 401 Unauthorized

Response body:

```
{ "message": "Authentication problems" }
```

HTTP Status code: 404 Not Found

Response body:

```
{ "message": "Service query errors" }
```

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Network issues" }
```

Services Management - Toggle Service

Method: POST

Path: DashboardNotifier.toggleService()

Purpose: Start or stop a service

Expected parameters: JSON format

```
{
  "serviceName": "SMB",
  "action": "start"
}
```

Output settings: JSON format

Success

HTTP Status code: 200 OK

Response body:

```
{
  "message": "Service toggled successfully",
  "serviceName": "SMB",
  "isRunning": true
}
```

Failures

HTTP Status code: 400 Bad Request

Response body:

```
{ "message": "Service not found" }
```

HTTP Status code: 401 Unauthorized

Response body:

```
{ "message": "Authentication problems" }
```

HTTP Status code: 403 Forbidden

Response body:

```
{ "message": "Permission denied" }
```

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Service operation error, Network connectivity
issues" }
```

Real-time Data - Subscribe to Metrics

Method: STREAM

Path: DashboardNotifier.getRealtimeSystemInfoStreamClean()

Purpose: Subscribe to real-time system metrics via WebSocket stream

Expected parameters: No input expected

Output settings: JSON event stream

Success

HTTP Status code: 200 OK (stream open)

Response body:

```
{
  "event": "metrics",
  "cpuUsagePct": 15.2,
  "memoryUsagePct": 64.0,
  "timestamp": "2025-09-26T12:00:10Z"
}
```

Failures

HTTP Status code: 401 Unauthorized

Response body:

```
{ "message": "Authentication problems" }
```

HTTP Status code: 404 Not Found

Response body:

```
{ "message": "WebSocket connection lost" }
```

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Subscription errors, Network interruptions" }
```

Data Persistence - Save Server

Method: POST

Path: ServerRepository.addServer()

Purpose: Save server configuration

Expected parameters: JSON format

```
{
  "name": "MyServer",
  "host": "192.168.1.10",
  "port": 443,
  "ssl": true,
  "acceptSelfSigned": false
}
```

Output settings: JSON format

Success

HTTP Status code: 201 Created

Response body:

```
{
  "id": 101,
  "message": "Server saved",
  "readyForConnection": true
}
```

Failures

HTTP Status code: 400 Bad Request

Response body:

```
{ "message": "Database constraint violations" }
```

HTTP Status code: 409 Conflict

Response body:

```
{ "message": "Duplicate server entries" }
```

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Storage errors" }
```

Data Persistence - Get All Servers

Method: GET

Path: ServerRepository.getAllServers()

Purpose: Retrieve all configured servers

Expected parameters: No input expected

Output settings: JSON format

Success

HTTP Status code: 200 OK

Response body:

```
[
  { "id": 101, "name": "MyServer", "host": "192.168.1.10",
"port": 443, "ssl": true, "lastConnected": "2025-09-20T10:00:00Z"
},
  { "id": 102, "name": "LabNAS",   "host": "10.0.0.5",
"port": 443, "ssl": true, "lastConnected": "2025-09-18T09:12:00Z"
}
]
```

Failures

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Database access errors, Query execution problems" }
```

Secure Storage - Save API Key

Method: POST

Path: SecureStorageService.saveApiKey()

Purpose: Store API token securely

Expected parameters: JSON format

```
{
  "serverId": 101,
  "apiKey": "API_KEY_EXAMPLE"
}
```

Output settings: JSON format

Success

HTTP Status code: 200 OK

Response body:

```
{
  "message": "Token stored",
  "serverId": 101
}
```

Failures

HTTP Status code: 400 Bad Request

Response body:

```
{ "message": "Invalid parameters" }
```

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Secure storage unavailable, Encryption errors,
Platform-specific issues" }
```

Secure Storage - Get API Key

Method: GET

Path: SecureStorageService.getApiKey()

Purpose: Retrieve stored API token

Expected parameters: JSON format

```
{
  "serverId": 101
}
```

Output settings: JSON format

Success

HTTP Status code: 200 OK

Response body:

```
{
  "serverId": 101,
  "apiKey": "API_KEY_EXAMPLE"
}
```

Failures

HTTP Status code: 404 Not Found

Response body:

```
{ "message": "Token not found" }
```

HTTP Status code: 500 Internal Server Error

Response body:

```
{ "message": "Secure storage errors, Decryption problems" }
```

5. Plan SCM and QA Strategies

SCM & QA Strategy Document

SCM Strategy (Source Code Management)

Version Control Tool

- Git as the primary version control system

- Repository hosted on GitHub for collaboration and backup

- Commit messages following conventional commits format (feat:, fix:, docs:, etc.)

Branching Strategy

- main: Production-ready code, protected branch

- develop: Integration branch for features, staging environment

- feature/: Feature branches for new functionality (e.g., feature/dashboard-realtime, feature/service-toggles)

- hotfix/: Emergency fixes for production issues

- release/: Release preparation branches

Development Workflow

- Feature branches created from develop

- Regular commits with descriptive messages

- Pull requests required for merging into develop

- Code reviews mandatory before merge approval

- Automated CI/CD checks on pull requests

- Merge to main only through release branches

Code Review Process

- Minimum 1 reviewer approval required

- Automated linting and formatting checks

- Unit test coverage requirements (minimum 80%)

- Security and performance review for critical changes

QA Strategy (Quality Assurance)

Testing Strategy

- Unit Tests: Individual component testing using Flutter's built-in testing framework

- Widget Tests: UI component testing for widgets and user interactions

- Integration Tests: End-to-end testing of complete user flows

- Manual Testing: Critical user journeys and edge cases

Testing Tools

- Flutter Test: Unit and widget testing framework

- Integration Test: End-to-end testing for complete app flows

- Mockito: Mocking framework for external dependencies

- Golden Tests: Visual regression testing for UI components

Test Coverage Requirements

- Unit tests: Minimum 80% code coverage

- Critical paths: 100% test coverage (connection, authentication, service toggles)

- Integration tests: All major user flows covered

Testing Types

- Functional Testing: Verify all features work as specified

- Performance Testing: App responsiveness and memory usage

- Security Testing: API token handling, secure storage validation

- Compatibility Testing: Different Android/iOS versions and screen sizes

- Network Testing: Offline scenarios, connection failures, timeout handling

Deployment Pipeline

Development

- Feature branches → develop branch

Staging

- Automated deployment from develop branch for testing

Production

- Release branches → main branch with manual approval

Automated Testing

- Unit tests, integration tests, and linting on every commit

Manual Testing

- Staging environment validation before production release

Quality Gates

- All tests must pass before merge

- Code coverage requirements met

- Security scan passed

- Performance benchmarks within acceptable limits

- Manual testing approval for critical features

Release Management

- Semantic versioning (MAJOR.MINOR.PATCH)

- Release notes documentation

- Rollback strategy for production issues

- Feature flags for gradual rollout of new functionality