# 📜 Amanita Wallet

## Общая идея

Amanita — это некастодиальный веб-кошелёк нового поколения, где главным приоритетом является простота и понятность для **low-tech пользователей**.
В классических криптокошельках основой является сид-фраза — длинный список слов, которые пользователь должен записать и хранить. На практике большинство людей теряют её, путаются, или хранят небезопасно. Это создаёт барьер входа и риск потери средств.

Мы предлагаем **обратный подход**:

- **Базовый, приоритетный способ**: доверенная передача «оберега» хранителю (социальное шифрование через кодовое слово).

- **Продвинутый, дополнительный способ**: сид-фраза (доступна только через специальную ссылку, для опытных пользователей).

Таким образом, вход в систему становится максимально дружелюбным, а безопасность — встроенной в социальные связи.

---

## Функциональные задачи

### Со стороны отправки (создания копии оберега)

1. Пользователь выбирает опцию **«Позвать хранителя»**.

2. Его просят придумать **секретное слово**, которое знают только он и доверенный человек.

3. Сид-фраза автоматически шифруется этим словом и превращается в **QR-оберег**.

4. Пользователь отправляет QR хранителю (например, в Telegram).

Задача пользователя — не хранить абстрактные слова, а всего лишь:

- помнить одно секретное слово,

- доверить QR близкому человеку.

---

### Со стороны восстановления доступа

1. Пользователь или хранитель открывает приложение и сканирует сохранённый QR-оберег.

2. Приложение понимает, что это зашифрованная сид-фраза.

3. Система просит ввести **секретное слово**.

4. После успешного ввода кошелёк восстанавливается автоматически.

Таким образом, восстановление становится **простым, социальным и бесшовным**: QR + слово.

---

## Концептуальный UX-стиль (синтез шаманизма и крипты)

Интерфейс Amanita должен ощущаться не как «технический инструмент для гиков», а как **ритуал взаимодействия с цифровым миром**.
Основные качества:

- **Сакральность вместо абстракции**: сид-фраза → «оберег», доверенное лицо → «хранитель».

- **Ритуальные действия вместо сложных инструкций**: «выберите секретное слово», «передайте QR хранителю».

- **Простота и символизм**: пользователь не тонет в терминах, а следует ясным шагам с образными пояснениями.

- **Нейтральная эстетика**: никакого кричащего UI или перегрузки визуальными деталями. Всё подаётся как **спокойный, уверенный интерфейс**, в котором цифровая криптография и шаманские метафоры объединяются в единый опыт.

### Визуально-интерфейсный синтез (без конкретики)

- **Формы** — плавные, органичные, с намёком на природные структуры (спираль, круг, ветвь).
- **Язык интерфейса** — метафорический, но ясный («оберег», «хранитель», «секретное слово»).
- **Механика** — линейный путь с короткими шагами, каждый шаг объясняется так, словно это маленький ритуал.
- **Иконография** — символическая (сундук, ключ, QR-талисман), а не техническая.

---

## Итог

В результате Amanita Wallet превращает **рискованную и путающую процедуру** хранения сид-фразы в **понятный и социально встроенный ритуал безопасности**.

- Для большинства пользователей: *QR-оберег + хранитель*.
- Для продвинутых: *сид-фраза по специальной ссылке*.

Так мы достигаем одновременно:

- простоты для low-tech профилей,
- сохранения децентрализации,
- уникального концептуального опыта на стыке **шаманизма и криптографии**.

---

## 🚩 Основная проблема без “Социального оберега”

- Если секретное слово хранится только в голове пользователя, то при его потере QR-оберег становится бесполезным.
- Low-tech аудитория часто **забывает пароли, путает слова, теряет записки**.
- В результате человек оказывается в той же ловушке, что и с сид-фразой → потеря доступа.

---

## ✅ Обоснование передачи секретного слова хранителю

1. **Социальный дубль безопасности**
    - Хранитель получает не только QR-оберег, но и *ключ к нему*.
    - Если пользователь всё забудет, у хранителя всё ещё останется «второй шанс» восстановить.
2. **Сохранение децентрализации**
    - Хранитель — это не сервер и не облако, а человек из реальной жизни.
    - Мы уходим от зависимости от централизованных провайдеров, но сохраняем надёжность через социальное доверие.
3. **Психологический комфорт**
    - Для low-tech человека идея «мама хранит секрет» проще и надёжнее, чем абстрактная «фраза из 12 слов».
    - Это естественный паттерн: в кризисе люди всегда обращаются к близким.
4. **Минимизация когнитивной нагрузки**
    - Пользователю не нужно помнить ничего сложного.
    - Достаточно знать: «Мой хранитель всё помнит».

---

## ⚖️ Баланс доверия и риска

- Мы вводим ограничение: хранитель должен быть человеком, которому доверяешь безусловно (мама, брат, партнёр).
- Даже если хранитель знает секретное слово, **он не сможет ничего сделать без самого QR-оберега** (и наоборот).
- То есть безопасность строится на **разделении ключа и замка**:

- QR-оберег = замок.
- Секретное слово = ключ.
- Только вместе они работают.

---

## ✨ Шаманская метафора

В терминах концептуального синтеза:

- Оберег без слова → это закрытый сундук.
- Слово без оберега → это ключ без двери.
- Только когда оба находятся в руках хранителя, оберег оживает и возвращает силу владельцу.

---

Таким образом, передача секретного слова хранителю — это не “слабое звено”, а **ритуальное удвоение защиты**:

- QR и слово разделены,
- пользователь и хранитель связаны доверием,
- восстановление становится возможным даже в случае человеческой ошибки.

# 📲 Assignment for React Native Developer: Amanita Wallet

## Context

We are building **Amanita Wallet** — a **non-custodial Web3 wallet** with a **shamanic UX profile**.
The main design decision:

- For **low-tech users** the **primary onboarding method** is *“Call the Guardian”* flow (QR-Obereg + secret word).

- Advanced users will still have access to the **classic seed phrase backup** (hidden behind a special link).

So the **core requirement** is: build a technical skeleton that ensures **maximum simplicity and maximum security** for end users.

---

# Phase 1 — Onboarding & Security (current scope)

## Core tasks

1. **Seed phrase generation**
    - Generate a standard BIP-39 mnemonic (12/24 words).
    - Store it only locally (never leave the device).
2. **Guardian Flow (main method)**
    - User chooses a **secret word** (string).
    - System encrypts the seed phrase with that secret word (AES-256 or equivalent).
    - Encrypted seed is transformed into a **QR code**.
    - User can export this QR (send via Telegram, save as image, etc.).
3. **Restoration**
    - User or Guardian scans the QR.
    - App detects it as encrypted seed phrase.
    - App asks for the secret word.
    - If correct → seamlessly restore wallet with all tokens/accounts.
4. **Advanced access (special link)**
    - For experienced users only: direct access to seed phrase (copy/save).
    - This should be hidden, require explicit action, and display warnings.

---

## Phase 2 — Wallet Functionality (sketch for later)

- Standard non-custodial wallet features.
- Manage multiple tokens:
    - **Personal tokens** (private use).
    - **Social tokens** (community/guardian/relationship layer).
    - **Business tokens** (loyalty points and e-commerce).
    - Many more DeFi for real economy and token-marketing will be coming, architecture must be flexible to welcome easily new features!!!
- Simple import flow for external tokens/accounts.
- Clear separation in the UI between categories (social, business, personal).

---

## Technical skeleton requirements

1. **Security first**
    - Never store unencrypted seed phrase or private keys outside secure storage.
    - Encryption/decryption must always happen on-device.
    - Secret word must not be logged or cached.
2. **Cross-platform**
    - React Native base (iOS + Android).
    - Consider integration with secure storage modules (`react-native-keychain`, `expo-secure-store`).
3. **Modularity**
    - Wallet core logic (keys, encryption, signing) must be cleanly separated from UI.

    - Guardian flow should be a self-contained module → easy to expand/change later.

4. **Extensibility**
    - Architecture must allow integration of token modules (social, business, personal).
    - Token handling layer must be abstracted → easy to add ERC-20/ERC-721 or custom token logic.
5. **User Experience**
    - Each step should be linear, minimal cognitive load.
    - Wording must stay in **"shamanic UX language"** (Guardian, Obereg, Secret Word) at the top level.
    - Internally, code must map these terms to standard crypto operations.

---

# Deliverables (initial)

- A documented **technical skeleton** of the app.
- Proposal for libraries & dependencies (encryption, QR generation, secure storage).
- Draft flow diagrams for:
    - Seed generation.
    - Guardian flow (QR + secret word).
    - Restoration.
    - Advanced seed export.

---

👉 **Goal**: You must design this skeleton so that the app can grow into a **full wallet** later, but already at the onboarding stage it solves the **low-tech + security paradox** with the Guardian Obereg flow.

# Technical Architecture

## 🔨🤖🔧 Phase 1: Onboarding & Security — Technical Skeleton

---

### 1️⃣ CORE MODULES & LAYERS (Top-Level Architecture)

```
/core
  /wallet     → BIP-39 generation, key derivation, restoration
  /crypto     → Encryption (AES-256), decryption, key derivation
from secret word
  /qr         → QR encode/decode utilities
  /storage    → Secure local storage abstraction

/modules
  /guardian   → Guardian flow: encryption, QR, restoration logic
  /advanced   → Advanced user features (export, seed reveal)

/ui
  /screens    → Onboarding, Guardian, Restoration, Advanced
  /components → Reusable visual components
  /hooks      → App logic (state, async, storage)

@types        → Type definitions: Seed, EncryptedSeed, Token, etc.
```

---

## 2️⃣ MODULE DECOMPOSITION

### 🌱 Seed Phrase (core/wallet)

- **Gen**: BIP-39 12/24 word mnemonic generation (e.g., `bip39`, `@scure/bip39`)
- **Derive**: Generate seed (BIP-32 root) + keypairs (later in Phase 2)
- **Local Storage**: Only store **encrypted** (never plaintext)
- **Pseudonym Mapping**:
    - "Seed Phrase" → internally: `mnemonic`

- "Secret Word" → internally: `passphrase`

### 🔐 Guardian Flow (modules/guardian)

- **Encrypt**:
  - AES-256 using key derived from secret word (`PBKDF2` or `scrypt`)
  - Transform encrypted blob into base64 or binary
- **Encode**: Turn encrypted blob into QR (`react-native-qrcode-svg`)
- **Decode + Decrypt**:
  - Scan QR
  - Prompt secret word
  - Derive key → decrypt → get original mnemonic

❗ Secret word is never stored, cached, or logged.

### 📦 Advanced Access (modules/advanced)

- Hidden behind an “Elder’s Path” link
- Multiple warnings, confirm dialogs
- One-time reveal of raw seed phrase
- Optional: download as file or copy to clipboard

---

# 3️⃣ TECHNICAL PRINCIPLES

### 🛡️ Security Principles

- **Zero Trust Local Storage**: Use `expo-secure-store` or `react-native-keychain`
- **Encryption**:

    - AES-GCM with 256-bit key
    - Use `crypto-js`, `libsodium`, or `react-native-simple-crypto`
- **Key Derivation**:
    - PBKDF2 with salt OR scrypt (more resistant to brute force)

### 📱 Platform

- **Base**: React Native + Expo (ejectable for native modules)
- **Secure Storage**:
    - Expo Secure Store (default)
    - react-native-keychain (optional native fallback)
- **QR**: `react-native-qrcode-svg` + `react-native-camera` / `expo-barcode-scanner`

---

## 4️⃣ GROWTH-READY STRUCTURE

### 🔌 Modular Token Handling (Future-Proofing)

- `/tokens`
    - ERC-20 / ERC-721 handler interfaces
    - Categories: social, business, personal
    - Easy registration of new token types

### 🔄 Async Engine (Future)

- Background sync manager
- Token indexer
- Wallet event listeners (subscriptions, token events)

---

## 5️⃣ UX FLOW DIAGRAMS — HIGH LEVEL

### 🔄 Seed Generation (Internal only)

```
Generate Mnemonic (12/24 words) → Store Temporarily in Memory
→ (if Guardian Flow) → Encrypt with Secret Word
→ Encode as QR → Export
```

### 🧙 Guardian Flow

```
User Chooses Secret Word → Encrypt Seed → Encode to QR
→ Show QR → Export Options (Telegram, Save, Share)
```

### 🧬 Restoration

```
Scan QR → Recognize Guardian Format → Prompt for Secret Word
→ Derive Key → Decrypt → Validate Mnemonic → Restore Wallet
```

### 🛡️ Advanced Access

```
User clicks “Advanced Access” → Warning Dialogs
→ Reveal Mnemonic → Copy / Save Option
```

---

## 6️⃣ MODULE RELATIONSHIPS

```
[ UI ] ⟷ [ modules/guardian ] ⟷ [ core/crypto, core/wallet,
core/qr, core/storage ]
        ↑                          ↑
[ modules/advanced ]              |
        ↑                          |
      (same core access, stricter UI handling)
```

---

## 🧭 Guiding UX Principles

- Linear steps (no backtracking unless needed)

- Guardian terminology always shown (“Obereg”, “Guardian”, “Secret Word”)
- Crypto terms are **hidden**, mapped internally
- Default path = Guardian
- Advanced path = Hidden

---

## ✅ Next Steps (Deliverables Roadmap)

1. Define types and interfaces (Seed, EncryptedSeed, SecretWord)
2. Choose and evaluate AES encryption libs for React Native
3. Build a `guardian-engine.ts` — handles encrypt/decrypt + QR generation
4. Prototype flow in mock screens
5. Confirm QR + Decryption works cross-device

## ✔️ Предложенная архитектура совместима с DeFi-экосистемой и может быть естественно расширена для реализации:

### 🔮 Бартерной экономики

- Торговля/обмен токенами без участия фиата или DEX.
- Прямая передача "ценности" между пользователями (p2p-соглашения).
- Интеграция с микромаркетплейсами внутри кошелька.

### 🎯 Лояльти экономики

- Привязка бизнес токенов к действиям: покупки, посещения, репутация.
- Эмиссия токенов "на лету" через внешние API или внутри кошелька.
- UI как "игровая панель" → понятная логика получения и траты.

---

## 💡 Как это можно встроить в текущую архитектуру:

### 1️⃣ Токен-движок (`/tokens`):

Создать ядро с **абстрактным токен-интерфейсом**, которое:

- понимает стандарты ERC-20, ERC-721, ERC-1155;
- может обрабатывать кастомные правила: эмиссия, сжигание, бартер, миссии.

```
interface TokenModule {
  id: string;
  category: 'personal' | 'social' | 'business';
  fetchBalance(address: string): Promise<BigNumber>;
  send(to: string, amount: BigNumber): Promise<string>;
  getMetadata(): Promise<TokenMeta>;
}
```

- Любую "игру" можно реализовать как модуль, реализующий этот интерфейс.

---

### 2️⃣ Категории токенов

Уже заложены: `personal`, `social`, `business` → отлично ложится на геймификацию и DeFi.

Примеры:

| Категория | Реальное применение | Возможный гейм-дизайн |
|---|---|---|
| `social` | токены доверия, кармы | «обряды», «обеты», «друзья» |
| `business` | лояльти токены магазинов | «дары», «скидки», «обмен дарами» |
| `personal` | коллекции, репутация, краудфандинг | «трофеи», «награды», «магии» |

---

### 3️⃣ "Игры" как надстройки над токен-модулем

Создаем слой `engines` или `games`, где каждая игра — это UI + логика взаимодействия с токеном:

```
/games
  /gift_exchange
  /karma_ritual
  /merchant_loyalty
```

Каждая игра:

- получает доступ к токенам через модуль `/tokens`
- отрисовывается в UX как самостоятельный ритуал, миссия или опыт

---

### 4️⃣ Экосистема будущего

- Интеграция с оффчейн-сервисами через Oracle/API (например, магазины).
- Локальные DeFi механики: staking, bartering, missions.
- DAO-механики (на уровне social tokens).

---

## ✅ Вывод

> Да, архитектура полностью совместима и уже **по духу** ориентирована на расширение под DeFi + игрофицированную реальную экономику.

### Сильные стороны:

- модульность
- четкое разделение токенов по контекстам
- расширяемость "игровыми" механиками
- сохранение безопасности и автономности