ReachMe.io Clone

🧩 Table of Contents

---

1. Overview

A decentralized messaging/paywall platform where users (followers) pay in **USDC on Base** to message KOLs (key opinion leaders). If the KOL responds within a timeframe, funds are released. Otherwise, the user can request a refund.

---

2. Core Features

👤 User Features

- Wallet login (SIWE)
- Custom profile page (reachme.io/username)
- Add contact methods
- Set pricing (USDC amount per message)
- Set auto-expiry for message refund window (e.g. 72h)
- View inbox and sent messages
- Manage USDC balance, withdrawals

🛠️ Admin Features

- User search, ban/unban
- View transactions/logs
- Enforce platform-level pricing caps
- Moderate abusive content
- Manual refund override

---

3. Architecture

User (Browser or Base Wallet MiniApp)
↓
Next.js Frontend (React + Tailwind)
↓
Next.js API Routes (Backend)
↓
PostgreSQL (Prisma ORM)
↓
Base Blockchain (USDC Payments via Smart Contract)

Tech Stack

Layer	Tech
Frontend	Next.js, TailwindCSS
Backend	Next.js API routes
Auth	Sign-In With Ethereum (SIWE)
DB	PostgreSQL via Prisma
Blockchain	Base chain with USDC
Smart Contracts	Escrow contract for paid messaging
Hosting	Docker / Kubernetes
Mini App	Follows Base Wallet App Spec

---

4. User Flows

4.1 User Registration

1. User connects their wallet.

2. SIWE nonce is requested → user signs.
3. Backend verifies signature → session is stored.
4. User sets:
    - Username
    - Profile picture / bio
    - Pricing per message
    - Refund window (e.g. 72h)

Relevant APIs

POST /api/auth/nonce
POST /api/auth/verify
GET /api/users/me
POST /api/users

4.2 Message Sending + Payment

1. Follower visits `/username`.
2. Fills message form.
3. Frontend initiates:
    - `sendMessage(recipient, messageText, feeAmount)` on-chain call (to escrow)
4. Message is stored off-chain in DB (linked to payment tx).
5. KOL gets notification.

Contract Behavior

- Validates correct `feeAmount`
- Escrows USDC
- Emits `MessageSent` event

Relevant APIs

POST /api/messages/send (requires tx hash)
GET /api/messages/sent

4.3 KOL Reply & Refund Flow

1. KOL reads message in their dashboard.
2. KOL clicks "Reply" (off-chain reply + mark as responded).
3. Backend triggers smart contract `releaseFunds(messageId)`.

If KOL does NOT reply in time:

4. After expiry (e.g. 72h), sender can call `claimRefund(messageId)`.
5. USDC is returned to the sender.

Edge Cases

- Refunds are only enabled after expiry timestamp.
- Admin can force a refund or payment.

---

5. Smart Contract Design

💡 Main Features

- Escrow pattern with:
    - `sendMessage(recipient, messageId, expiry)`
    - `releaseFunds(messageId)` (by KOL)
    - `claimRefund(messageId)` (by sender after expiry)

- USDC-only (ERC20)
- Emits events for front-end sync
- Implement OpenZeppelin's ReentrancyGuard on all external-facing functions OpenzeppelinOpenzeppelin and use the checks-effects-interactions pattern throughout your contract architecture
- Use block numbers instead of timestamps for precise timing, or implement tolerance ranges to account for manipulation.
- Must account for USDC's blacklist functionality and potential depeg scenarios.

✅ Solidity (Simplified)

```
contract MessageEscrow {
   IERC20 public usdc;

   struct Message {
      address sender;
      address recipient;
      uint256 amount;
      uint256 expiry;
      bool responded;
      bool refunded;
   }

   mapping(bytes32 => Message) public messages;
```

```
    constructor(address _usdc) {
        usdc = IERC20(_usdc);
    }

    function sendMessage(
        address recipient,
        bytes32 messageId,
        uint256 amount,
        uint256 expiry
    ) external {
        require(usdc.transferFrom(msg.sender, address(this), amount), "USDC failed");
        messages[messageId] = Message(msg.sender, recipient, amount, expiry, false, false);
        emit MessageSent(messageId, msg.sender, recipient, amount, expiry);
    }

    function releaseFunds(bytes32 messageId) external {
        Message storage msgData = messages[messageId];
        require(msg.sender == msgData.recipient, "Not recipient");
        require(!msgData.responded && !msgData.refunded, "Already handled");

        msgData.responded = true;
        require(usdc.transfer(msgData.recipient, msgData.amount), "USDC fail");
        emit FundsReleased(messageId);
    }

    function claimRefund(bytes32 messageId) external {
        Message storage msgData = messages[messageId];
        require(msg.sender == msgData.sender, "Not sender");
        require(block.timestamp > msgData.expiry, "Too early");
        require(!msgData.responded && !msgData.refunded, "Already handled");

        msgData.refunded = true;
        require(usdc.transfer(msgData.sender, msgData.amount), "USDC fail");
        emit Refunded(messageId);
    }

    event MessageSent(bytes32 indexed id, address sender, address recipient, uint256
amount, uint256 expiry);
    event FundsReleased(bytes32 indexed id);
    event Refunded(bytes32 indexed id);
}
```

---

6. API Design

🔐 Authentication

Endpoint	Description
`POST /api/auth/nonce`	Get SIWE nonce
`POST /api/auth/verify`	Verify wallet signature
`GET /api/auth/me`	Get current user
`POST /api/auth/logout`	End session

👤 Users

Endpoint	Description
`GET /api/users/:username`	Public profile
`POST /api/users`	Create or update profile
`GET /api/users/me`	Private profile info

📬 Messages

Endpoint	Description
`POST /api/messages/send`	Submit message with payment hash
`GET /api/messages/inbox`	KOL's inbox
`GET /api/messages/sent`	Sender history
`POST /api/messages/reply`	Mark as replied
`POST /api/messages/refund`	Trigger refund (post expiry)

🛠️ Admin

Endpoint	Description
`GET /api/admin/users`	List users
`POST /api/admin/users/ban`	Ban user
`GET /api/admin/logs`	View message/payment logs
`POST /api/admin/override-refund`	Force refund
`POST /api/admin/remove-message`	Delete message

---

7. Admin Features

- Protected by wallet login + admin flag
- Global search by username or wallet
- Transaction viewer per user
- Abuse reporting (flag messages)
  Manual refund or payment override
  Account status toggling (active/banned)

---

8. Deployment & Integration

🧪 Environments

- `.env.local` for:
  - `DATABASE_URL`
  - `BASE_RPC_URL`
  - `USDC_CONTRACT`
  - `ESCROW_CONTRACT`
- Use `hardhat` or `foundry` for deployment

☁️ Deployment Options

- Frontend: Docker
- Backend: Same as frontend (Next.js API)
- DB: PostgreSQL

- Contracts: Deployed on Base Mainnet and Testnet

---

9. MiniApp Wallet Compatibility

- Must follow Base Mini App Spec
- Responsive design
- No popups or new tabs
- Support deep linking with `base://` URLs
- Optimized for in-app browser UX
- mobile-first design is non-negotiable
- Progressive Web App functionality for app-like experience