«هویت مرموز»

مستند طراحی محصول پیام‌رسان غیرمتمرکز

به سفارش «مانی‌لب»
استودیو بلاکچین از ۱۳۹۲

ویرایش نخست - ۶ خرداد ۱۴۰۴
میرسهیل نیک‌زاد کلورزی
با همکاری جمینای گوگل

فهرست عناوین

فهرست عناوین 2
مقدمه و نمای کلی سیستم ((System Overview 4
۱.۱. معرفی محصول: 4
۱.۲. اهداف کلیدی: 4
۱.۳. معماری سیستم: 4
۱.۴. جریان کلی کارکرد (:Overall Workflow) 5
راه‌اندازی پروژه: 5
اتصال کیف پول (:Connect Wallet) 5
راه‌اندازی :XMTP Client 5
ارسال پیام/فایل: 5
دریافت پیام: 5
منوها و رابط کاربری ((UI/UX Specification 5
۲.۱. ساختار :HTML (DOM Structure) 5
۲.۲. جریان‌های :UI/UX (User Flows) 6
۲.۲.۱. جریان اتصال کیف پول و راه‌اندازی :XMTP 6
۲.۲.۲. جریان ارسال پیام: 6
۲.۲.۳. جریان ارسال فایل: 7
۲.۲.۴. جریان دریافت و نمایش پیام: 7
جزئیات فنی و توابع ((Technical Details & Functions 8
۳.۱. متغیرهای سراسری (:Global Variables & State Management) 8
۳.۲. وابستگی‌های package.json :NPM (Package Dependencies) 8
۳.۳. توابع اصلی (:Core Functions) 9
۳.۳.۱. :isMetaMaskInstalled() 9
۳.۳.۲. :connectWalletBtn.addEventListener('click', async () => { ... }) 9
۳.۳.۳. :initXmtpBtn.addEventListener('click', async () => { ... }) 9
۳.۳.۴. :sendMessageBtn.addEventListener('click', async () => { ... }) 9
۳.۳.۵. :sendFileBtn.addEventListener('click', async () => { ... }) 10
۳.۳.۶. :displayMessage(message) 11
پکیج‌ها و سناریوی کارکرد ((Packages & Operational Scenario 11
۴.۱. جزئیات پکیج‌ها (:Package Specifics) 11
۴.۱.۱. :ethers.js 11
۴.۱.۲. :@xmtp/xmtp-js 11
۴.۲. سناریوی کارکرد (:Operational Scenario / User Journey) 12
۴.۲.۱. سناریو ۱: کاربری جدید برای DeChat و XMTP 12
۴.۲.۲. سناریو ۲: ارسال فایل 13
۴.۲.۳. سناریو ۳: تغییر محیط XMTP 13
ملاحظات امنیتی، ملاحظات عملکردی، و توسعه آتی ((Security, Performance & Future Development 13
۵.۱. ملاحظات امنیتی (:Security Considerations) 13
۵.۱.۱. ریسک‌ها و راهکارهای کاهش آن‌ها: 13
۵.۲. ملاحظات عملکردی (:Performance Considerations) 14
۵.۳. توسعه آتی (:Future Development) 14

نتیجه‌گیری: 14
نسخه انگلیسی جهت پرامپت 15
Product Design Document: Decentralized XMTP Messenger (DeChat) 15
Page 1: Introduction and System Overview 15
Page 2: UI/UX Specification 16
Page 3: Technical Details and Functions 19
Page 4: Packages and Operational Scenario 23
Page 5: Security, Performance, and Future Development 25

مقدمه و نمای کلی سیستم (System Overview)

۱.۱. معرفی محصول:

"پیام‌رسان غیرمتمرکز XMTP" (که از این پس به اختصار DeChat نامیده می‌شود) یک (Decentralized Chat Application (DApp است که ارتباطات کاربر-به-کاربر را بر بستر پروتکل XMTP (Extensible Message Transport Protocol) و با استفاده از کیف پول‌های مبتنی بر اتریوم (مانند MetaMask) فراهم می‌کند. این اپلیکیشن با هدف ارائه یک بستر پیام‌رسانی امن، مقاوم در برابر سانسور، و متعلق به کاربر، طراحی شده است که در آن، مالکیت داده‌ها و هویت ارتباطی به جای سرورهای متمرکز، در اختیار کاربران است.

۱.۲. اهداف کلیدی:

- پیام‌رسانی Peer-to-Peer: امکان ارسال و دریافت پیام‌های متنی و پیوست‌ها (فایل‌ها) بین کاربران با آدرس‌های کیف پول اتریوم.
- هویت غیرمتمرکز: استفاده از آدرس کیف پول به عنوان شناسه اصلی کاربر، بدون نیاز به ثبت‌نام یا حساب کاربری سنتی.
- امنیت ارتباطات: رمزنگاری End-to-End تمامی پیام‌ها از طریق پروتکل XMTP.
- کاربری ساده: ارائه رابط کاربری بصری و حداقل پیچیدگی برای تعامل با پروتکل‌های Web3.
- انعطاف‌پذیری محیط: امکان انتخاب محیط‌های XMTP (Dev/Production) برای توسعه و استقرار.

۱.۳. معماری سیستم:

System Architecture): DeChat) یک (Single-Page Application (SPA توسعه یافته با Vanilla JavaScript است که از یک ابزار باندلینگ مدرن (Vite) برای مدیریت وابستگی‌ها و بهینه‌سازی فرآیند توسعه استفاده می‌کند. معماری آن به شرح زیر است:

- Front-end:
  - HTML5: ساختار اصلی رابط کاربری.
  - CSS3: استایل‌دهی و طراحی بصری.
  - Vanilla JavaScript (ES Modules): منطق برنامه، تعامل با UI، و فراخوانی APIهای Web3.
- Web3 Libraries (Installed via npm/yarn):
  - ethers.js (v6+): برای تعامل با کیف پول MetaMask، امضای تراکنش‌ها و مدیریت ارائه‌دهنده اتریوم (Provider).
  - @xmtp/xmtp-js (v11+): پیاده‌سازی پروتکل XMTP برای ارسال/دریافت پیام‌ها، مدیریت مکالمات، و رمزنگاری.
- Development & Build Tool:
  - Vite: یک ابزار باندلینگ سریع که قابلیت‌هایی مانند HMR (Hot Module Replacement) و بهینه‌سازی کد برای Production را فراهم می‌کند. Vite از npm یا yarn برای مدیریت پکیج‌ها استفاده می‌کند و import های ES Module را به درستی resolve می‌کند.
- Runtime Environment:
  - Web Browser: کروم (با افزونه MetaMask)، فایرفاکس (با افزونه MetaMask)، یا هر مرورگر Web3-enabled.
  - MetaMask: کیف پول اتریوم برای احراز هویت و امضای پیام‌ها/تراکنش‌ها.

۱.۴. جریان کلی کارکرد (Overall Workflow):

راه‌اندازی پروژه:

توسعه‌دهنده پروژه را با Node.js و Vite راه‌اندازی می‌کند.

اتصال کیف پول (Connect Wallet):

کاربر با کلیک بر روی دکمه "اتصال کیف پول"، اتصال MetaMask خود را آغاز می‌کند. ethers.js از window.ethereum برای این منظور استفاده می‌کند.

راه‌اندازی XMTP Client:

پس از اتصال موفق کیف پول، کاربر دکمه "راه‌اندازی XMTP" را کلیک می‌کند. xmtp-js با استفاده از signer (امضاکننده) از ethers، یک کلاینت XMTP را با انتخاب محیط (Dev/Production) ایجاد می‌کند.

ارسال پیام/فایل:

کاربر آدرس گیرنده را وارد کرده، پیام یا فایل را انتخاب و ارسال می‌کند. xmtp-js پیام را رمزنگاری کرده و از طریق شبکه XMTP ارسال می‌کند.

دریافت پیام:

کلاینت XMTP به طور مداوم به دنبال پیام‌های جدید در مکالمات فعال گوش می‌دهد و پیام‌های دریافتی را رمزگشایی و در UI نمایش می‌دهد.

منوها و رابط کاربری (UI/UX Specification)

۲.۱. ساختار HTML (DOM Structure):

رابط کاربری در فایل index.html تعریف شده و شامل بخش‌های اصلی زیر است:

- div.container: کانتینر اصلی برنامه.
  - h1: عنوان اصلی "پیام‌رسان غیرمتمرکز XMTP".
  - div.status-area#status: نمایش پیام‌های عمومی وضعیت برنامه (مثلاً "متصل شوید به کیف پول").
  - div.connection-status#connectionStatus: نمایش وضعیت اتصال کیف پول و آدرس آن.
  - div.controls: شامل دکمه‌های اصلی تعامل:
    - button#connectWalletBtn: اتصال کیف پول.
    - button#disconnectWalletBtn: قطع اتصال کیف پول (در ابتدا غیرفعال).
    - button#initXmtpBtn: راه‌اندازی XMTP Client (در ابتدا غیرفعال).
  - div.messaging-section (با display: none; در ابتدا): بخش اصلی پیام‌رسانی که پس از راه‌اندازی موفق XMTP نمایش داده می‌شود.
    - div.section-header: "ارسال پیام".
    - div.messaging-status#messagingStatus: نمایش وضعیت مربوط به عملیات پیام‌رسانی (مثلاً "در حال ارسال پیام...").
    - input#recipientAddress: فیلد متنی برای ورود آدرس کیف پول گیرنده.
    - textarea#messageInput: فیلد متنی برای ورود محتوای پیام.
    - button#sendMessageBtn: دکمه ارسال پیام.
    - div.file-upload-section: بخش ارسال فایل.

- div.section-header: "ارسال فایل".
- input#fileInput: ورودی نوع فایل برای انتخاب فایل.
- button#sendFileBtn: دکمه ارسال فایل.
- div.file-status#fileStatus: نمایش وضعیت ارسال فایل (مثلاً "فایل با موفقیت ارسال شد.").
- div.section-header: "گفتگو".
- div.chat-area#chatArea: ناحیه نمایش پیام‌ها (مکالمه).
- div.section-header: "انتخاب محیط".
- select#envSelect: لیست کشویی برای انتخاب محیط XMTP (Dev/Production).
- button#setEnvBtn: دکمه اعمال محیط (برای راه‌اندازی مجدد XMTP Client با محیط جدید).

۲.۲. جریان‌های UI/UX (User Flows):

۲.۲.۱. جریان اتصال کیف پول و راه‌اندازی XMTP:

1. شروع: صفحه با وضعیت اولیه "برای شروع، دکمه 'اتصال کیف پول' را فشار دهید." بارگذاری می‌شود. دکمه‌های "قطع اتصال" و "راه‌اندازی XMTP" غیرفعال هستند. بخش پیام‌رسانی (messaging-section.) پنهان است.
2. کلیک بر "اتصال کیف پول" (connectWalletBtn):
   - اگر MetaMask نصب نباشد: statusDiv پیام خطا نمایش می‌دهد.
   - اگر MetaMask نصب باشد: درخواست eth_requestAccounts به window.ethereum ارسال می‌شود.
   - موفقیت:
     - signer از ethers.BrowserProvider دریافت می‌شود.
     - statusDiv و connectionStatusDiv آدرس کیف پول متصل شده را نمایش می‌دهند.
     - connectWalletBtn غیرفعال، disconnectWalletBtn و initXmtpBtn فعال می‌شوند.
   - خطا: statusDiv پیام خطا را نمایش می‌دهد.
3. کلیک بر "راه‌اندازی XMTP" (initXmtpBtn):
   - messagingStatusDiv پیام "در حال راه‌اندازی XMTP..." را نمایش می‌دهد.
   - xmtp.Client.create با signer و env انتخاب شده فراخوانی می‌شود.
   - موفقیت:
     - xmtpClient ایجاد می‌شود.
     - messagingStatusDiv آدرس XMTP و محیط را نمایش می‌دهد.
     - messagingSection (بخش پیام‌رسانی) قابل مشاهده می‌شود.
     - initXmtpBtn غیرفعال می‌شود.
   - خطا: messagingStatusDiv پیام خطا را نمایش می‌دهد.
4. کلیک بر "قطع اتصال" (disconnectWalletBtn):
   - تمام متغیرهای مربوط به اتصال (signer, xmtpClient, conversation) به null بازنشانی می‌شوند.
   - UI به وضعیت اولیه باز می‌گردد: دکمه‌های "اتصال کیف پول" فعال، سایر دکمه‌ها غیرفعال، بخش پیام‌رسانی پنهان می‌شود.

۲.۲.۲. جریان ارسال پیام:

1. ورود اطلاعات: کاربر آدرس گیرنده را در recipientAddressInput و پیام را در messageInput وارد می‌کند.
2. کلیک بر "ارسال پیام" (sendMessageBtn):
   - اگر xmtpClient راه‌اندازی نشده باشد: messagingStatusDiv پیام خطا نمایش می‌دهد.
   - اگر آدرس گیرنده یا محتوای پیام خالی باشد: messagingStatusDiv پیام خطا نمایش می‌دهد.

- در حال ارسال: messagingStatusDiv پیام "در حال ارسال پیام..." را نمایش می‌دهد.
- ارسال:
  - xmtpClient.conversations.newConversation(recipientAddress) فراخوانی می‌شود.
  - conversation.send(messageContent) فراخوانی می‌شود.
  - موفقیت:
    - messagingStatusDiv پیام "پیام با موفقیت ارسال شد." را نمایش می‌دهد.
    - پیام ارسال شده به صورت محلی در chatArea نمایش داده می‌شود (با کلاس self).
    - messageInput پاک می‌شود.
    - شروع استریم پیام: پس از ارسال اولین پیام به یک مکالمه جدید، conversation.streamMessages() فعال می‌شود تا پیام‌های جدید ورودی برای آن مکالمه را دریافت کند.
  - خطا: messagingStatusDiv پیام خطا را نمایش می‌دهد.

۲.۲.۳. جریان ارسال فایل:

1. ورود اطلاعات: کاربر آدرس گیرنده را در recipientAddressInput وارد می‌کند و فایل را از طریق fileInput انتخاب می‌کند.
2. کلیک بر "ارسال فایل" (sendFileBtn):
   - اگر xmtpClient راه‌اندازی نشده باشد: fileStatusDiv پیام خطا نمایش می‌دهد.
   - اگر آدرس گیرنده یا فایل انتخاب نشده باشد: fileStatusDiv پیام خطا نمایش می‌دهد.
   - در حال ارسال: fileStatusDiv پیام "در حال آماده‌سازی و ارسال فایل..." را نمایش می‌دهد و نمایش داده می‌شود.
   - ارسال:
     - فایل با FileReader به ArrayBuffer تبدیل می‌شود و سپس به Blob تبدیل می‌شود.
     - یک آبجکت attachment شامل filename, mimeType, و content (Blob) ساخته می‌شود.
     - xmtpClient.conversations.newConversation(recipientAddress) فراخوانی می‌شود.
     - conversation.send(attachment, { contentType: ContentTypes.Attachment }) فراخوانی می‌شود.
     - موفقیت:
       - fileStatusDiv پیام "فایل با موفقیت ارسال شد." را نمایش می‌دهد.
       - یک پیام نماینده (مثلاً "[پیوست: نام‌فایل]") به صورت محلی در chatArea نمایش داده می‌شود.
       - fileInput پاک می‌شود.
     - خطا: fileStatusDiv پیام خطا را نمایش می‌دهد.

۲.۲.۴. جریان دریافت و نمایش پیام:

- هنگامی که conversation.streamMessages() فعال است، به صورت ناهمزمان به پیام‌های جدید گوش می‌دهد.
- هر پیام جدیدی که از طریق استریم دریافت می‌شود (و پیام خود کاربر نباشد)، توسط تابع displayMessage در chatArea نمایش داده می‌شود.
- پیام‌های دریافتی با کلاس other در chatArea نمایش داده می‌شوند.

جزئیات فنی و توابع (Technical Details & Functions)

۳.۱. متغیرهای سراسری (Global Variables & State Management):

متغیرهای کلیدی برای حفظ وضعیت برنامه در سراسر main.js تعریف شده‌اند:

- signer: نوع ethers.Signer. نمایانگر حساب اتریوم متصل شده و قابلیت امضا دارد. پس از اتصال کیف پول مقدار می‌گیرد.
- xmtpClient: نوع xmtp.Client. نمونه کلاینت XMTP که برای تعامل با پروتکل استفاده می‌شود. پس از راه‌اندازی XMTP مقدار می‌گیرد.
- conversation: نوع xmtp.Conversation. شیء گفتگوی فعال فعلی (بین کاربر و گیرنده). پس از ایجاد/انتخاب گفتگو مقدار می‌گیرد.

۳.۲. وابستگی‌های package.json :NPM (Package Dependencies)

در ریشه پروژه، وابستگی‌های کلیدی را تعریف می‌کند:

JSON

```
}
  "name": "my-xmtp-app",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "devDependencies": {
    "vite": "^5.2.0"
  },
  "dependencies": {
    "@xmtp/xmtp-js": "^11.2.0", // آخرین نسخه پایدار xmtp-js
    "ethers": "^6.12.1"       // آخرین نسخه پایدار ethers v6
  }
}
```

نکات فنی:

- "type": "module": این تنظیم در package.json به Node.js و Vite می‌گوید که فایل‌های جاوااسکریپت در این پروژه از سینتکس ES Module (یعنی import/export) استفاده می‌کنند.
- devDependencies: شامل ابزارهای توسعه مانند Vite.
- dependencies: شامل پکیج‌های اصلی برنامه مانند ethers و @xmtp/xmtp-js.

۳.۳. توابع اصلی (Core Functions):

۳.۳.۱. isMetaMaskInstalled():

- هدف: بررسی وجود افزونه MetaMask در مرورگر.

پیاده‌سازی:

JavaScript

function isMetaMaskInstalled() {
return typeof window.ethereum !== 'undefined';
}

-
- نکات فنی: window.ethereum یک شیء سراسری است که توسط MetaMask تزریق می‌شود و دروازه اصلی تعامل با بلاکچین اتریوم از طریق مرورگر است.

۳.۳.۲. connectWalletBtn.addEventListener('click', async () => { ... }):

- هدف: مدیریت فرآیند اتصال کیف پول MetaMask.
- فرایند فنی:
  1. بررسی MetaMask: فراخوانی isMetaMaskInstalled().
  2. درخواست حساب‌ها: استفاده از window.ethereum.request({ method: 'eth_requestAccounts' }). این متد یک پنجره پاپ‌آپ MetaMask را برای درخواست مجوز دسترسی به حساب‌های کاربر باز می‌کند.
  3. ایجاد Signer:
     - const provider = new BrowserProvider(window.ethereum);: ایجاد یک BrowserProvider از ethers.js که به window.ethereum متصل می‌شود. BrowserProvider به طور خاص برای محیط‌های مرورگر طراحی شده است.
     - const signer = await provider.getSigner();: دریافت signer از provider. signer یک آبجکت است که قادر به امضای پیام‌ها و تراکنش‌ها با کلید خصوصی کاربر است (بدون افشای کلید خصوصی).
- مدیریت خطا: هرگونه خطا در فرآیند (مثل رد کردن اتصال توسط کاربر) در بلوک catch مدیریت و در statusDiv نمایش داده می‌شود.

۳.۳.۳. initXmtpBtn.addEventListener('click', async () => { ... }):

- هدف: راه‌اندازی کلاینت XMTP.
- فرایند فنی:
  1. پیش‌شرط: بررسی وجود signer.
  2. انتخاب محیط: const env = envSelect.value; برای انتخاب بین dev و production برای شبکه XMTP. محیط dev برای توسعه و آزمایش مناسب است.
  3. ایجاد کلاینت XMTP:
     - const xmtpClient = await Client.create(signer, { env });: این مهمترین گام است. xmtp-js از signer برای تولید یک privateKeyBundle استفاده می‌کند که برای رمزنگاری پیام‌ها و احراز هویت در شبکه XMTP لازم است. این privateKeyBundle به صورت خودکار توسط XMTP ایجاد یا از Local Storage بازیابی می‌شود.
- مدیریت خطا: خطاها در بلوک catch مدیریت می‌شوند.

۳.۳.۴. sendMessageBtn.addEventListener('click', async () => { ... }):

- هدف: ارسال پیام متنی.

- فرایند فنی:
  1. اعتبار سنجی ورودی‌ها: بررسی آدرس گیرنده و محتوای پیام.
  2. ایجاد/دریافت مکالمه:
     - ;(conversation = await xmtpClient.conversations.newConversation(recipientAddress: این متد یک مکالمه جدید با آدرس گیرنده ایجاد می‌کند یا اگر مکالمه‌ای از قبل وجود دارد، آن را بازیابی می‌کند. conversation یک شیء است که شامل متدها و خصوصیات مربوط به آن مکالمه خاص است.
  3. ارسال پیام:
     - ;(await conversation.send(messageContent: پیام متنی را به صورت رمزنگاری شده از طریق شبکه XMTP ارسال می‌کند.
  4. نمایش محلی: فراخوانی displayMessage برای نمایش فوری پیام ارسال شده در UI (با کلاس self).
  5. استریم پیام‌های ورودی:
     - ;() { conversation.stream.return } (conversation && conversation.stream)if: این خط بسیار مهم است. اگر قبلاً یک استریم برای این مکالمه وجود داشته، آن را خاتمه می‌دهد تا از ایجاد استریم‌های تکراری و تداخل جلوگیری شود.
     - ;() { ... } (()for await (const message of await conversation.streamMessages: این یک حلقه ناهمزمان (async iterator) است که به طور مداوم به دنبال پیام‌های جدید در این مکالمه گوش می‌دهد. هر زمان که یک پیام جدید دریافت شود، بدنه حلقه اجرا می‌شود.
     - ;{ continue } (message.senderAddress === xmtpClient.address)if: جلوگیری از نمایش دو باره پیام‌های خودی (یک بار هنگام ارسال، یک بار از طریق استریم).
     - ;(displayMessage(message: نمایش پیام دریافتی در UI (با کلاس other).
- مدیریت خطا: خطاها در بلوک catch مدیریت می‌شوند.

۳.۳.۵. sendFileBtn.addEventListener('click', async () => { ... }):

- هدف: ارسال فایل به عنوان پیوست.
- فرایند فنی:
  1. اعتبار سنجی ورودی‌ها: بررسی آدرس گیرنده و انتخاب فایل.
  2. خواندن فایل:
     - ;()const reader = new FileReader: استفاده از FileReader API مرورگر برای خواندن محتوای فایل.
     - ;(reader.readAsArrayBuffer(file: خواندن فایل به عنوان ArrayBuffer.
     - ;{ ... } => () reader.onloadend = async: کد ارسال فایل پس از اتمام خواندن فایل اجرا می‌شود.
     - ;({ const blob = new Blob([arrayBuffer], { type: file.type: تبدیل ArrayBuffer به Blob که برای xmtp-js لازم است.
  3. آماده‌سازی پیوست:
     - ;{ const attachment = { filename: file.name, mimeType: file.type, content: blob: ایجاد یک آبجکت attachment با فرمت مورد نیاز xmtp-js.
  4. ارسال پیوست:
     - ;({ await conversation.send(attachment, { contentType: ContentTypes.Attachment: ارسال پیوست. نکته کلیدی اینجاست که از ContentTypes.Attachment (که از @xmtp/xmtp-js وارد شده) به عنوان contentType پیام استفاده می‌شود تا پروتکل بداند که این یک پیام پیوست است.
- مدیریت خطا: خطاها در بلوک catch مدیریت می‌شوند.

۳.۳.۶. displayMessage(message):

- هدف: نمایش پیام‌ها در UI (chatArea).
- فرایند فنی:
  1. ایجاد عنصر HTML: یک div جدید ایجاد می‌شود.
  2. تعیین فرستنده:
     - messageDiv.classList.add(message.senderAddress === (xmtpClient ? 'xmtpClient.address : null) ? 'self' : 'other');: بر اساس آدرس فرستنده پیام و آدرس کاربر فعلی، کلاس CSS self یا other به پیام اختصاص داده می‌شود تا استایل‌دهی متفاوت (مثل پیام‌های خودی در سمت راست، پیام‌های دیگران در سمت چپ) اعمال شود.
  3. پردازش محتوا:
     - بررسی message.contentType.id === ContentTypes.Attachment.id و ContentTypes.RemoteAttachment.id برای نمایش مناسب پیوست‌ها (به جای نمایش محتوای باینری، یک متن نماینده مانند "[پیوست: نام‌فایل]" نمایش داده می‌شود).
  4. اضافه کردن به DOM: پیام به chatArea اضافه می‌شود.
  5. اسکرول: chatArea.scrollTop = chatArea.scrollHeight; برای اطمینان از اینکه جدیدترین پیام همیشه در دید باشد.

پکیج‌ها و سناریوی کارکرد (Packages & Operational Scenario)

۴.۱. جزئیات پکیج‌ها (Package Specifics):

۴.۱.۱. ethers.js:

- نقش: کتابخانه اصلی برای تعامل با بلاکچین اتریوم.
- ماژول‌های مورد استفاده:
  - BrowserProvider: یک کلاس Provider برای تعامل با ارائه‌دهندگان (Provider) تزریق شده توسط مرورگر (مانند MetaMask).
  - Signer: یک آبجکت انتزاعی که نمایانگر یک حساب اتریوم است و می‌تواند تراکنش‌ها و پیام‌ها را امضا کند.
- نحوه استفاده:
  - new BrowserProvider(window.ethereum): ایجاد یک اتصال به MetaMask.
  - provider.getSigner(): دریافت امضاکننده فعال (حساب متصل شده).
  - این کتابخانه ضروری برای احراز هویت و اتصال به شبکه XMTP است، زیرا XMTP از امضاهای اتریوم برای تأیید هویت و رمزنگاری استفاده می‌کند.

۴.۱.۲. @xmtp/xmtp-js:

- نقش: پیاده‌سازی جاوااسکریپت پروتکل XMTP. این کتابخانه تمامی منطق رمزنگاری، ذخیره‌سازی، و مسیریابی پیام‌ها را برعهده دارد.
- ماژول‌های مورد استفاده:
  1. Client: کلاس اصلی برای ایجاد و مدیریت ارتباط با شبکه XMTP. این کلاس شامل متدهایی برای ایجاد کلاینت، مدیریت مکالمات، و ارسال/دریافت پیام‌ها است.
     - Client.create(signer, { env: 'dev' | 'production' }): متد استاتیک برای ایجاد یک نمونه کلاینت XMTP. به signer از ethers برای احراز هویت نیاز دارد.

2. ContentTypes: یک آبجکت که شامل تعاریف انواع محتوای پشتیبانی شده توسط XMTP است (مانند ContentTypes.Attachment, ContentTypes.Text). این‌ها برای مشخص کردن نوع داده‌ای که ارسال می‌شود استفاده می‌شوند.

- مدل داده اصلی:
  1. Client: نماینده کاربر در شبکه XMTP.
  2. Conversation: نمایانگر یک گفتگوی دو نفره بین دو آدرس XMTP. می‌توان آن را با xmtpClient.conversations.newConversation(peerAddress) ایجاد یا بازیابی کرد.
  3. Message: شیء پیام که شامل senderAddress, content (محتوای رمزگشایی شده), sentAt (زمان ارسال), و contentType است.
- جریان پیام‌رسانی داخلی XMTP (خلاصه):
  1. Initialization: کاربر با کیف پول خود امضا می‌کند تا privateKeyBundle تولید شود.
  2. Key Exchange (بسته به حالت): اگر مکالمه جدید باشد، کلیدهای رمزنگاری بین دو طرف تبادل می‌شوند.
  3. Encryption: پیام با استفاده از کلیدهای مشترک و الگوریتم‌های رمزنگاری End-to-End (مشابه OMEMO/Signal Protocol) رمزنگاری می‌شود.
  4. Publishing: پیام رمزنگاری شده به شبکه XMTP (روی IPFS یا زیرساخت‌های مرتبط) منتشر می‌شود.
  5. Retrieval: کلاینت گیرنده پیام‌های جدید را از شبکه پایش می‌کند، آن‌ها را بازیابی و با استفاده از کلیدهای خود رمزگشایی می‌کند.

۴.۲. سناریوی کارکرد (Operational Scenario / User Journey):

۴.۲.۱. سناریو ۱: کاربری جدید برای DeChat و XMTP

1. کاربر: "من می‌خواهم برای اولین بار از این پیام‌رسان استفاده کنم."
2. DeChat UI: صفحه را نمایش می‌دهد با دکمه "اتصال کیف پول" فعال.
3. کاربر: "روی دکمه 'اتصال کیف پول' کلیک می‌کند."
4. MetaMask: پنجره پاپ‌آپ MetaMask باز می‌شود و از کاربر می‌خواهد کیف پول خود را انتخاب و تأیید کند.
5. کاربر: کیف پول خود را انتخاب و "تأیید" می‌کند.
6. DeChat UI: وضعیت اتصال کیف پول را نمایش می‌دهد. دکمه "راه‌اندازی XMTP" فعال می‌شود.
7. کاربر: "روی دکمه 'راه‌اندازی XMTP' کلیک می‌کند."
8. XMTP Client: (در پس‌زمینه) یک XMTP "Enable Identity" درخواست می‌کند که MetaMask آن را به عنوان یک "Sign Message" نمایش می‌دهد. این امضا برای تولید privateKeyBundle XMTP لازم است و هیچ هزینه گس (Gas Fee) ندارد.
9. کاربر: امضا را در MetaMask تأیید می‌کند.
10. DeChat UI: پیام "XMTP راه‌اندازی شد" را نمایش می‌دهد. بخش پیام‌رسانی فعال و قابل استفاده می‌شود.
11. کاربر: آدرس یک دوست (مثلاً 0xabc...123) را در فیلد گیرنده وارد می‌کند.
12. کاربر: پیامی ("سلام دوست من!") را در messageInput تایپ و "ارسال پیام" را کلیک می‌کند.
13. DeChat UI: پیام "در حال ارسال پیام..." را نمایش می‌دهد.
14. XMTP Client: یک مکالمه جدید با 0xabc...123 ایجاد می‌کند. پیام را رمزنگاری کرده و به شبکه XMTP ارسال می‌کند.
15. DeChat UI: پیام "پیام با موفقیت ارسال شد." را نمایش می‌دهد و پیام ارسال شده را به صورت محلی در چت نمایش می‌دهد.
16. XMTP Client: استریم پیام‌ها را برای این مکالمه آغاز می‌کند و در انتظار پیام‌های جدید از 0xabc...123 می‌ماند.
17. دوست کاربر: پیام را دریافت می‌کند و پاسخ می‌دهد ("سلام! پیام شما رسید.").
18. DeChat UI: پیام دوست کاربر را از طریق استریم دریافت، رمزگشایی و در چت نمایش می‌دهد.

۴.۲.۲. سناریو ۲: ارسال فایل

1. پیش‌فرض: کاربر قبلاً کیف پول خود را متصل کرده و XMTP را راه‌اندازی کرده است.
2. کاربر: آدرس گیرنده را در فیلد مربوطه وارد می‌کند.
3. کاربر: روی input#fileInput کلیک کرده و یک فایل (مثلاً یک عکس) از سیستم خود انتخاب می‌کند.
4. کاربر: "روی دکمه 'ارسال فایل' کلیک می‌کند."
5. DeChat UI: پیام "در حال آماده‌سازی و ارسال فایل..." را در fileStatusDiv نمایش می‌دهد.
6. XMTP Client: فایل را می‌خواند، آن را به Blob تبدیل می‌کند و به عنوان یک Attachment با contentType: ContentTypes.Attachment از طریق پروتکل XMTP ارسال می‌کند.
7. DeChat UI: پیام "فایل با موفقیت ارسال شد." را نمایش می‌دهد و یک نشانگر ("پیوست: نام‌فایل") را در چت نمایش می‌دهد.

۴.۲.۳. سناریو ۳: تغییر محیط XMTP

1. پیش‌فرض: کاربر قبلاً کیف پول خود را متصل کرده و XMTP را راه‌اندازی کرده است.
2. کاربر: از select#envSelect گزینه "Production" را انتخاب می‌کند.
3. کاربر: "روی دکمه 'اعمال محیط' کلیک می‌کند."
4. DeChat UI: initXmtpBtn را فعال می‌کند و یک کلیک مجازی روی آن انجام می‌دهد.
5. XMTP Client: دوباره راه‌اندازی می‌شود (با یک درخواست امضای مجدد برای تولید privateKeyBundle در محیط جدید، اگر قبلاً ایجاد نشده باشد) و به شبکه Production XMTP متصل می‌شود.
6. DeChat UI: پیام "XMTP راه‌اندازی شد. (محیط: Production)" را نمایش می‌دهد.

ملاحظات امنیتی، ملاحظات عملکردی، و توسعه آتی (Security, Performance & Future Development)

۵.۱. ملاحظات امنیتی (Security Considerations):

- رمزنگاری End-to-End: هسته اصلی امنیت XMTP است. تمامی پیام‌ها بین فرستنده و گیرنده رمزنگاری می‌شوند و حتی سرورهای XMTP نیز قادر به خواندن محتوای آن‌ها نیستند.
- عدم نیاز به سرور واسطه برای کلید خصوصی: کلید خصوصی کاربر هرگز از کیف پول خارج نمی‌شود. امضای پیام‌ها/تراکنش‌ها توسط افزونه MetaMask در مرورگر انجام می‌شود.
- مقاومت در برابر سانسور: از آنجایی که پیام‌ها بر بستر شبکه‌های غیرمتمرکز (مانند IPFS) ذخیره می‌شوند، حذف یا سانسور آن‌ها توسط یک نهاد مرکزی دشوار است.
- امنیت Wallet Connect: فرآیند اتصال به کیف پول از طریق window.ethereum و ethers.js کاملاً استاندارد و امن است و متکی بر پروتکل Wallet Connect (یا مشابه آن در MetaMask) است.
- مسائل احراز هویت: XMTP از امضای Cryptographic برای تأیید هویت استفاده می‌کند، بنابراین اطمینان از اینکه آدرس کیف پول متعلق به فرستنده است، بالا است.

۵.۱.۱. ریسک‌ها و راهکارهای کاهش آن‌ها:

- فیشینگ (Phishing): کاربران باید همیشه مراقب URL وب‌سایت باشند و از اتصال کیف پول خود به وب‌سایت‌های مشکوک خودداری کنند. (این یک ریسک عمومی Web3 است که به DApp محدود نمی‌شود.)
- امنیت کیف پول: اگر کیف پول کاربر (MetaMask) به خطر بیفتد، کلید خصوصی XMTP نیز ممکن است به خطر بیفتد. آموزش کاربران در مورد نگهداری امن از عبارات بازیابی (seed phrase) و کلیدهای خصوصی ضروری است.

- آدرس‌های نامعتبر: کاربران باید از صحت آدرس گیرنده اطمینان حاصل کنند، زیرا پیام‌ها به آدرس‌های اشتباه، قابل برگشت نیستند.

۵.۲. ملاحظات عملکردی (Performance Considerations):

- بارگذاری اولیه: با استفاده از Vite، فرآیند باندلینگ و مینی‌فای کردن کد انجام می‌شود که حجم فایل‌های جاوااسکریپت را برای بارگذاری اولیه بهینه می‌کند.
- عملکرد XMTP Client:
  - ایجاد کلاینت: xmtp.Client.create ممکن است بسته به اینکه آیا privateKeyBundle قبلاً تولید و ذخیره شده باشد، کمی زمان‌بر باشد.
  - جستجو/استریم پیام‌ها: استریم پیام‌ها به صورت Real-time انجام می‌شود و عملکرد آن به سرعت شبکه XMTP و اینترنت کاربر بستگی دارد.
- پردازش فایل: خواندن و تبدیل فایل‌ها در سمت کلاینت با FileReader انجام می‌شود و عملکرد آن به اندازه فایل و قدرت پردازشی دستگاه کاربر بستگی دارد.
- مصرف منابع مرورگر: حفظ استریم پیام‌ها در طولانی مدت ممکن است مصرف منابع مرورگر را کمی افزایش دهد. در یک برنامه پیچیده‌تر، مدیریت صحیح استریم‌ها (قطع و وصل کردن هوشمندانه) برای بهینه‌سازی منابع ضروری است.

۵.۳. توسعه آتی (Future Development):

- لیست گفتگوها: افزودن قابلیت نمایش لیست تمامی گفتگوهای کاربر و انتخاب آن‌ها برای ادامه چت (با استفاده از xmtpClient.conversations.list()).
- وضعیت آنلاین/آفلاین: پیاده‌سازی مکانیزمی برای نمایش وضعیت آنلاین/آفلاین کاربران (اگر XMTP این قابلیت را فراهم کند یا با استفاده از پروتکل‌های جانبی).
- اعلان‌ها (Notifications): پیاده‌سازی اعلان‌های مرورگر برای پیام‌های جدید دریافتی، حتی زمانی که برنامه در پس‌زمینه است.
- پشتیبانی از فرمت‌های محتوای بیشتر: گسترش پشتیبانی از انواع محتوای XMTP (مانند Reply, Reaction، یا GroupChat در صورت اضافه شدن به XMTP).
- مدیریت گروه‌ها: در صورت توسعه قابلیت‌های گروه‌های چت در XMTP، افزودن پشتیبانی از چت‌های گروهی.
- نمایش وضعیت خوانده شدن (Read Receipts): پیاده‌سازی قابلیت نشان دادن زمان خوانده شدن پیام توسط گیرنده.
- ذخیره‌سازی محلی (Caching): استفاده از IndexedDB یا سایر مکانیزم‌های ذخیره‌سازی محلی برای بارگذاری سریع‌تر تاریخچه پیام‌ها.
- تنظیمات پیشرفته: افزودن تنظیمات برای کاربر، مانند انتخاب کیفیت ارسال فایل، یا مدیریت کلیدها.
- فشرده‌سازی فایل: پیاده‌سازی فشرده‌سازی فایل قبل از ارسال برای بهینه‌سازی مصرف پهنای باند و سرعت ارسال.

نتیجه‌گیری:

این مستند طراحی، یک پایه فنی قوی برای توسعه پیام‌رسان غیرمتمرکز DeChat بر اساس پروتکل XMTP فراهم می‌کند. با پیروی از این ساختار و استفاده از ابزارهای مدرن مانند Vite، می‌توان یک محصول پایدار، امن و قابل مقیاس‌بندی ساخت. موفقیت در این پروژه، نیازمند دقت در پیاده‌سازی جزئیات فنی و همچنین درک عمیق از ماهیت غیرمتمرکز پروتکل XMTP و ویژگی‌های آن است.

نسخه انگلیسی جهت پرامپت

Product Design Document: Decentralized XMTP Messenger (DeChat)

Date: May 27, 2025 **Version:** 1.0.0 **Author:** Gemini (under direct user guidance)

Page 1: Introduction and System Overview

1.1. **Product Introduction:** The "Decentralized XMTP Messenger" (hereafter referred to as **DeChat**) is a Decentralized Chat Application (DApp) designed to facilitate peer-to-peer communication over the Extensible Message Transport Protocol (XMTP) using Ethereum-based wallets such as MetaMask. The application aims to provide a secure, censorship-resistant, and user-owned messaging platform where data and communication identity reside with the users, rather than centralized servers.

1.2. **Key Objectives:**

- **Peer-to-Peer Messaging:** Enable sending and receiving of text messages and attachments (files) between users identified by their Ethereum wallet addresses.
- **Decentralized Identity:** Utilize the Ethereum wallet address as the primary user identifier, eliminating the need for traditional registration or account creation.
- **Secure Communications:** Ensure End-to-End Encryption (E2EE) for all messages via the XMTP protocol.
- **User-Friendly Interface:** Offer an intuitive user interface with minimal complexity for interacting with Web3 protocols.
- **Environment Flexibility:** Allow selection between XMTP's Dev and Production environments for development and deployment purposes.

1.3. **System Architecture:** DeChat is a Single-Page Application (SPA) developed with Vanilla JavaScript, leveraging a modern bundling tool (Vite) for dependency management and optimized development workflow. Its architecture is structured as follows:

- **Front-end:**
  - **HTML5:** Defines the core structure of the user interface.
  - **CSS3:** Provides styling and visual design.
  - **Vanilla JavaScript (ES Modules):** Handles application logic, UI interactions, and Web3 API calls.
- **Web3 Libraries (Installed via npm/yarn):**

- **`ethers.js`** (v6+): Utilized for interacting with the MetaMask wallet, signing transactions, and managing the Ethereum provider.
- **`@xmtp/xmtp-js`** (v11+): Implements the XMTP protocol for sending/receiving messages, managing conversations, and handling encryption.

- Development & Build Tool:
  - Vite: A fast bundling tool offering features like Hot Module Replacement (HMR) and optimized code for production builds. Vite utilizes `npm` or `yarn` for package management and correctly resolves ES Module imports.
- Runtime Environment:
  - Web Browser: Chrome (with MetaMask extension), Firefox (with MetaMask extension), or any Web3-enabled browser.
  - MetaMask: The Ethereum wallet serving as the authentication and signing mechanism for messages/transactions.

1.4. Overall Workflow:

1. Project Setup: The developer sets up the project using Node.js and Vite.
2. Wallet Connection: The user initiates the connection to their MetaMask wallet by clicking the "Connect Wallet" button. `ethers.js` leverages `window.ethereum` for this purpose.
3. XMTP Client Initialization: After a successful wallet connection, the user clicks the "Initialize XMTP" button. `xmtp-js` creates an XMTP client using the `signer` obtained from `ethers`, with an option to select the desired environment (Dev/Production).
4. Message/File Sending: The user inputs a recipient address, types a message, or selects a file, then initiates sending. `xmtp-js` encrypts the message/file content and dispatches it over the XMTP network.
5. Message Reception: The XMTP client continuously listens for new messages in active conversations, decrypts the received messages, and displays them in the UI.

<div style="page-break-after: always;"></div>

Page 2: UI/UX Specification

2.1. HTML Structure (DOM Structure): The user interface is defined within the `index.html` file and comprises the following primary sections:

- **`div.container`**: The main application container.
  - **`h1`**: Main title, "Decentralized XMTP Messenger".
  - **`div.status-area#status`**: Displays general application status messages (e.g., "Connect your wallet to start.").
  - **`div.connection-status#connectionStatus`**: Shows the wallet connection status and connected address.
  - **`div.controls`**: Contains primary interaction buttons:

- `button#connectWalletBtn`: Initiates wallet connection.
- `button#disconnectWalletBtn`: Disconnects the wallet (initially disabled).
- `button#initXmtpBtn`: Initializes the XMTP Client (initially disabled).

- **`div.messaging-section`** (initially **`display: none;`**): The core messaging area, which becomes visible after successful XMTP initialization.
  - **`div.section-header`**: "Send Message".
  - **`div.messaging-status#messagingStatus`**: Displays status messages related to messaging operations (e.g., "Sending message...").
  - `input#recipientAddress`: Text input field for entering the recipient's wallet address.
  - `textarea#messageInput`: Text area for composing the message content.
  - `button#sendMessageBtn`: Button to send the message.
  - **`div.file-upload-section`**: File sending area.
    - `div.section-header`: "Send File".
    - `input#fileInput`: File input element for selecting a file.
    - `button#sendFileBtn`: Button to send the selected file.
    - `div.file-status#fileStatus`: Displays file sending status (e.g., "File sent successfully.").
  - **`div.section-header`**: "Conversation".
  - **`div.chat-area#chatArea`**: The display area for messages (chat history).
  - **`div.section-header`**: "Select Environment".
  - `select#envSelect`: Dropdown for selecting the XMTP environment (Dev/Production).
  - `button#setEnvBtn`: Button to apply the selected environment (triggers XMTP Client re-initialization).

2.2. UI/UX Flows (User Flows):

2.2.1. Wallet Connection and XMTP Initialization Flow:

1. Start: The page loads with the initial status message "To start, press the 'Connect Wallet' button." The "Disconnect Wallet" and "Initialize XMTP" buttons are disabled. The messaging section (`.messaging-section`) is hidden.
2. Click "Connect Wallet" (**`connectWalletBtn`**):
   - If MetaMask is not installed: `statusDiv` displays an error message.
   - If MetaMask is installed: An `eth_requestAccounts` request is sent to `window.ethereum`.
   - Success:
     - A `signer` is obtained from `ethers.BrowserProvider`.
     - `statusDiv` and `connectionStatusDiv` display the connected wallet address.
     - `connectWalletBtn` becomes disabled, while `disconnectWalletBtn` and `initXmtpBtn` become enabled.
   - Error: `statusDiv` displays an error message.
3. Click "Initialize XMTP" (**`initXmtpBtn`**):

- `messagingStatusDiv` displays "Initializing XMTP...".
- `xmtp.Client.create` is invoked with the `signer` and the selected `env`.
- Success:
  - `xmtpClient` is instantiated.
  - `messagingStatusDiv` displays the XMTP address and environment.
  - `messagingSection` (the messaging area) becomes visible.
  - `initXmtpBtn` becomes disabled.
- Error: `messagingStatusDiv` displays an error message.

4. Click "Disconnect Wallet" (**`disconnectWalletBtn`**):
   - All connection-related variables (`signer`, `xmtpClient`, `conversation`) are reset to `null`.
   - The UI reverts to its initial state: "Connect Wallet" button enabled, others disabled, messaging section hidden.

2.2.2. Send Message Flow:

1. Input: The user enters the recipient's address in `recipientAddressInput` and the message in `messageInput`.
2. Click "Send Message" (**`sendMessageBtn`**):
   - If `xmtpClient` is not initialized: `messagingStatusDiv` displays an error.
   - If recipient address or message content is empty: `messagingStatusDiv` displays an error.
   - Sending: `messagingStatusDiv` displays "Sending message...".
   - Transmission:
     - `xmtpClient.conversations.newConversation(recipientAddress)` is called.
     - `conversation.send(messageContent)` is called.
     - Success:
       - `messagingStatusDiv` displays "Message sent successfully.".
       - The sent message is displayed locally in `chatArea` (with class `self`).
       - `messageInput` is cleared.
       - Message Stream Initiation: After the first message to a new conversation, `conversation.streamMessages()` is activated to receive incoming messages for that conversation.
     - Error: `messagingStatusDiv` displays an error message.

2.2.3. Send File Flow:

1. Input: The user enters the recipient's address in `recipientAddressInput` and selects a file via `fileInput`.
2. Click "Send File" (**`sendFileBtn`**):
   - If `xmtpClient` is not initialized: `fileStatusDiv` displays an error and becomes visible.
   - If recipient address or file is not selected: `fileStatusDiv` displays an error and becomes visible.
   - Sending: `fileStatusDiv` displays "Preparing and sending file..." and becomes visible.

- Transmission:
  - The file is read into an `ArrayBuffer` and then converted to a `Blob` using `FileReader`.
  - An `attachment` object containing `filename`, `mimeType`, and `content` (Blob) is constructed.
  - `xmtpClient.conversations.newConversation(recipientAddress)` is called.
  - `conversation.send(attachment, { contentType: ContentTypes.Attachment })` is called.
  - Success:
    - `fileStatusDiv` displays "File sent successfully.".
    - A placeholder message (e.g., "[Attachment: filename.ext]") is displayed locally in `chatArea`.
    - `fileInput` is cleared.
  - Error: `fileStatusDiv` displays an error message.

2.2.4. Receive and Display Message Flow:

- When `conversation.streamMessages()` is active, it asynchronously listens for new messages.
- Any new message received through the stream (and not sent by the current user) is processed by the `displayMessage` function and rendered in `chatArea`.
- Received messages are displayed in `chatArea` with the `other` class.

<div style="page-break-after: always;"></div>

Page 3: Technical Details and Functions

3.1. Global Variables & State Management: Key variables maintaining application state are defined globally within `main.js`:

- `signer`: Type `ethers.Signer`. Represents the connected Ethereum account with signing capabilities. It is assigned a value after wallet connection.
- `xmtpClient`: Type `xmtp.Client`. The XMTP client instance used for protocol interaction. It is assigned a value after XMTP initialization.
- `conversation`: Type `xmtp.Conversation`. The currently active conversation object (between the user and a specific recipient). It is assigned a value upon conversation creation/selection.

3.2. NPM Dependencies (Package Dependencies): The `package.json` file in the project root defines the key dependencies:

JSON
{

```
 "name": "my-xmtp-app",
 "private": true,
 "version": "0.0.0",
 "type": "module",
 "scripts": {
   "dev": "vite",
   "build": "vite build",
   "preview": "vite preview"
 },
 "devDependencies": {
   "vite": "^5.2.0"
 },
 "dependencies": {
   "@xmtp/xmtp-js": "^11.2.0", // Latest stable xmtp-js version
   "ethers": "^6.12.1"        // Latest stable ethers v6 version
 }
}
```

Technical Notes:

- `"type": "module"`: This setting in `package.json` instructs Node.js and Vite that JavaScript files in this project use ES Module syntax (`import`/`export`).
- `devDependencies`: Includes development tools like Vite.
- `dependencies`: Includes core application packages like `ethers` and `@xmtp/xmtp-js`.

3.3. Core Functions:

3.3.1. **`isMetaMaskInstalled()`**:

- **Purpose:** Checks for the presence of the MetaMask browser extension.

Implementation:

```
 JavaScript
function isMetaMaskInstalled() {
   return typeof window.ethereum !== 'undefined';
}
```

-
- **Technical Notes:** `window.ethereum` is a global object injected by MetaMask, serving as the primary gateway for browser-based Ethereum blockchain interaction.

3.3.2. **`connectWalletBtn.addEventListener('click', async () => { ... })`**:

- **Purpose:** Manages the MetaMask wallet connection process.

- Technical Process:
    1. MetaMask Check: Invokes `isMetaMaskInstalled()`.
    2. Account Request: Sends `window.ethereum.request({ method: 'eth_requestAccounts' })`. This method opens a MetaMask pop-up requesting user permission to access their accounts.
    3. Signer Creation:
        - `const provider = new BrowserProvider(window.ethereum);`: Creates an `ethers.js BrowserProvider` connected to `window.ethereum`. `BrowserProvider` is specifically designed for browser environments.
        - `signer = await provider.getSigner();`: Retrieves the `signer` from the `provider`. The `signer` is an object capable of signing messages and transactions with the user's private key (without exposing the private key).
- Error Handling: Any errors during the process (e.g., user rejecting connection) are caught in the `catch` block and displayed in `statusDiv`.

3.3.3. `initXmtpBtn.addEventListener('click', async () => { ... })`:

- Purpose: Initializes the XMTP client.
- Technical Process:
    1. Pre-condition: Checks for the existence of `signer`.
    2. Environment Selection: `const env = envSelect.value;` for selecting between `dev` and `production` XMTP networks. The `dev` environment is suitable for development and testing.
    3. XMTP Client Creation:
        - `xmtpClient = await Client.create(signer, { env });`: This is the crucial step. `xmtp-js` uses the `signer` to generate a `privateKeyBundle`, which is necessary for message encryption and authentication on the XMTP network. This `privateKeyBundle` is either generated automatically by XMTP or retrieved from Local Storage.
- Error Handling: Errors are managed in the `catch` block.

3.3.4. `sendMessageBtn.addEventListener('click', async () => { ... })`:

- Purpose: Sends a text message.
- Technical Process:
    1. Input Validation: Checks for valid recipient address and message content.
    2. Conversation Retrieval/Creation:
        - `conversation = await xmtpClient.conversations.newConversation(recipientAddress);`: This method either creates a new conversation with the recipient address or retrieves an existing one. The `conversation` object contains methods and properties related to that specific conversation.
    3. Message Sending:
        - `await conversation.send(messageContent);`: Sends the text message, encrypted, over the XMTP network.

4. Local Display: Calls `displayMessage` to immediately render the sent message in the UI (with class `self`).
5. Incoming Message Stream:
    - `if (conversation && conversation.stream) { conversation.stream.return(); }`: This line is vital. If a stream for this conversation already exists, it terminates it to prevent redundant streams and conflicts.
    - `for await (const message of await conversation.streamMessages()) { ... }`: This is an asynchronous iterator loop that continuously listens for new messages in this conversation. When a new message is received, the loop body executes.
    - `if (message.senderAddress === xmtpClient.address) { continue; }`: Prevents self-sent messages from being displayed twice (once on send, once from the stream).
    - `displayMessage(message);`: Displays the received message in the UI (with class `other`).

- Error Handling: Errors are managed in the `catch` block.

3.3.5. `sendFileBtn.addEventListener('click', async () => { ... })`:

- Purpose: Sends a file as an attachment.
- Technical Process:
    1. Input Validation: Checks for a valid recipient address and selected file.
    2. File Reading:
        - `const reader = new FileReader();`: Uses the browser's FileReader API to read the file content.
        - `reader.readAsArrayBuffer(file);`: Reads the file as an `ArrayBuffer`.
        - `reader.onloadend = async () => { ... };`: The file sending code executes after the file has been fully read.
        - `const blob = new Blob([arrayBuffer], { type: file.type });`: Converts the `ArrayBuffer` to a `Blob`, which is required by `xmtp-js`.
    3. Attachment Preparation:
        - `const attachment = { filename: file.name, mimeType: file.type, content: blob };`: Creates an `attachment` object in the format required by `xmtp-js`.
    4. Attachment Sending:
        - `await conversation.send(attachment, { contentType: ContentTypes.Attachment });`: Sends the attachment. The key here is using `ContentTypes.Attachment` (imported from `@xmtp/xmtp-js`) as the message `contentType` to instruct the protocol that this is an attachment message.
- Error Handling: Errors are managed in the `catch` block.

3.3.6. `displayMessage(message)`:

- Purpose: Renders messages in the `chatArea` (UI).

- Technical Process:
    1. HTML Element Creation: A new `div` element is created.
    2. Sender Determination:
        - `messageDiv.classList.add(message.senderAddress === (xmtpClient ? xmtpClient.address : null) ? 'self' : 'other');`: Based on the message sender's address and the current user's address, either the `self` or `other` CSS class is assigned to the message for distinct styling (e.g., self-messages on the right, others on the left).
    3. Content Processing:
        - Checks `message.contentType.id === ContentTypes.Attachment.id` and `ContentTypes.RemoteAttachment.id` for proper display of attachments (instead of binary content, a placeholder text like "[Attachment: filename.ext]" is shown).
    4. DOM Appendage: The message is appended to the `chatArea`.
    5. Scrolling: `chatArea.scrollTop = chatArea.scrollHeight;` ensures the latest message is always in view.

<div style="page-break-after: always;"></div>

Page 4: Packages and Operational Scenario

4.1. Package Specifics:

4.1.1. `ethers.js`:

- Role: The foundational library for interacting with the Ethereum blockchain.
- Modules Used:
    - `BrowserProvider`: A `Provider` class specifically designed for interacting with browser-injected providers (like MetaMask).
    - `Signer`: An abstract object representing an Ethereum account that can sign transactions and messages.
- Usage:
    - `new BrowserProvider(window.ethereum)`: Establishes a connection to MetaMask.
    - `provider.getSigner()`: Retrieves the currently active `signer` (the connected account).
    - This library is essential for authentication and connecting to the XMTP network, as XMTP leverages Ethereum signatures for identity verification and encryption.

4.1.2. `@xmtp/xmtp-js`:

- Role: The JavaScript implementation of the XMTP protocol. This library handles all the encryption, storage, and message routing logic.
- Modules Used:

1. `Client`: The main class for creating and managing a connection to the XMTP network. It includes methods for client creation, conversation management, and message sending/receiving.
    - `Client.create(signer, { env: 'dev' | 'production' })`: Static method to create an XMTP client instance. Requires a `signer` from `ethers` for authentication.
2. `ContentTypes`: An object containing definitions for various content types supported by XMTP (e.g., `ContentTypes.Attachment`, `ContentTypes.Text`). These are used to specify the type of data being sent.

- Core Data Model:
    1. **`Client`**: Represents the user's presence on the XMTP network.
    2. **`Conversation`**: Represents a two-person chat between two XMTP addresses. It can be created or retrieved using `xmtpClient.conversations.newConversation(peerAddress)`.
    3. **`Message`**: The message object, containing `senderAddress`, `content` (decrypted payload), `sentAt` (timestamp), and `contentType`.
- Internal XMTP Messaging Flow (Summary):
    1. **Initialization:** The user signs with their wallet to generate a `privateKeyBundle`.
    2. **Key Exchange (conditional):** If it's a new conversation, encryption keys are exchanged between the two parties.
    3. **Encryption:** The message is encrypted using shared keys and End-to-End encryption algorithms (similar to OMEMO/Signal Protocol).
    4. **Publishing:** The encrypted message is published to the XMTP network (on IPFS or related infrastructure).
    5. **Retrieval:** The recipient's client monitors the network for new messages, retrieves them, and decrypts them using their own keys.

4.2. Operational Scenario / User Journey:

4.2.1. Scenario 1: New User to DeChat and XMTP

1. **User:** "I want to use this messenger for the first time."
2. **DeChat UI:** Displays the page with the "Connect Wallet" button enabled.
3. **User:** Clicks the "Connect Wallet" button.
4. **MetaMask:** A MetaMask pop-up window appears, prompting the user to select and confirm their wallet.
5. **User:** Selects their wallet and clicks "Confirm."
6. **DeChat UI:** Displays the wallet connection status. The "Initialize XMTP" button becomes enabled.
7. **User:** Clicks the "Initialize XMTP" button.
8. **XMTP Client:** (In the background) XMTP requests an "Enable Identity" signature, which MetaMask presents as a "Sign Message" request. This signature is required to generate the XMTP `privateKeyBundle` and incurs no gas fees.
9. **User:** Confirms the signature in MetaMask.
10. **DeChat UI:** Displays "XMTP Initialized" message. The messaging section becomes active and usable.
11. **User:** Enters a friend's address (e.g., `0xabc...123`) in the recipient field.

12. **User:** Types a message ("Hello friend!") into `messageInput` and clicks "Send Message."
13. **DeChat UI:** Displays "Sending message..." status.
14. **XMTP Client:** Creates a new conversation with `0xabc...123`. Encrypts the message and sends it to the XMTP network.
15. **DeChat UI:** Displays "Message sent successfully." and shows the sent message locally in the chat.
16. **XMTP Client:** Initiates message streaming for this conversation, waiting for new messages from `0xabc...123`.
17. **User's Friend:** Receives the message and replies ("Hi! Got your message.").
18. **DeChat UI:** Receives the friend's message via the stream, decrypts it, and displays it in the chat.

4.2.2. Scenario 2: Sending a File

1. **Prerequisite:** The user has already connected their wallet and initialized XMTP.
2. **User:** Enters the recipient's address in the corresponding field.
3. **User:** Clicks on `input#fileInput` and selects a file (e.g., an image) from their system.
4. **User:** Clicks the "Send File" button.
5. **DeChat UI:** Displays "Preparing and sending file..." in `fileStatusDiv`, which becomes visible.
6. **XMTP Client:** Reads the file, converts it to a `Blob`, and sends it as an `Attachment` with `contentType: ContentTypes.Attachment` via the XMTP protocol.
7. **DeChat UI:** Displays "File sent successfully." and shows a placeholder ("[Attachment: filename.ext]") in the chat.

4.2.3. Scenario 3: Changing XMTP Environment

1. **Prerequisite:** The user has already connected their wallet and initialized XMTP.
2. **User:** Selects "Production" from the `select#envSelect` dropdown.
3. **User:** Clicks the "Apply Environment" button.
4. **DeChat UI:** Enables `initXmtpBtn` and virtually clicks it.
5. **XMTP Client:** Re-initializes (with another signature request for `privateKeyBundle` generation in the new environment, if not already created) and connects to the XMTP Production network.
6. **DeChat UI:** Displays "XMTP Initialized. (Environment: Production)" message.

<div style="page-break-after: always;"></div>

Page 5: Security, Performance, and Future Development

5.1. Security Considerations:

- **End-to-End Encryption (E2EE):** This is XMTP's core security feature. All messages are encrypted between sender and receiver, ensuring that even XMTP servers cannot read their content.
- **No Private Key Exposure:** The user's private key never leaves their wallet. Message/transaction signing is handled by the MetaMask extension within the browser.

- **Censorship Resistance:** Since messages are stored on decentralized networks (like IPFS), it is difficult for a central authority to delete or censor them.
- **Wallet Connect Security:** The wallet connection process via `window.ethereum` and `ethers.js` is standard and secure, relying on the Wallet Connect protocol (or similar in MetaMask).
- **Authentication:** XMTP uses cryptographic signatures for identity verification, ensuring a high degree of certainty that the wallet address belongs to the sender.

5.1.1. Risks and Mitigation Strategies:

- **Phishing:** Users must always verify website URLs and avoid connecting their wallets to suspicious websites. (This is a general Web3 risk not limited to the DApp.)
- **Wallet Security:** If the user's wallet (MetaMask) is compromised, their XMTP private key bundle may also be compromised. Educating users about secure handling of seed phrases and private keys is crucial.
- **Invalid Addresses:** Users must ensure the correctness of the recipient's address, as messages sent to incorrect addresses are irreversible.

5.2. Performance Considerations:

- **Initial Load:** By utilizing Vite, code bundling and minification occur, optimizing JavaScript file sizes for initial loading.
- **XMTP Client Performance:**
  - **Client Creation:** `xmtp.Client.create` might take some time, depending on whether the `privateKeyBundle` has been previously generated and stored.
  - **Message Fetching/Streaming:** Message streaming is real-time, and its performance depends on the XMTP network speed and the user's internet connection.
- **File Processing:** File reading and conversion are handled client-side using `FileReader`, and performance depends on file size and the user's device processing power.
- **Browser Resource Consumption:** Maintaining message streams over extended periods might slightly increase browser resource consumption. In more complex applications, intelligent stream management (smartly disconnecting/reconnecting) is essential for resource optimization.

5.3. Future Development:

- **Conversation List:** Adding the ability to display a list of all user conversations and select them to continue chatting (using `xmtpClient.conversations.list()`).
- **Online/Offline Status:** Implementing a mechanism to display user online/offline status (if XMTP provides this capability or through auxiliary protocols).
- **Notifications:** Implementing browser notifications for new incoming messages, even when the application is in the background.
- **Support for More Content Formats:** Expanding support for additional XMTP content types (e.g., `Reply`, `Reaction`, or `GroupChat` if added to XMTP).
- **Group Management:** If XMTP develops group chat functionalities, adding support for group chats.
- **Read Receipts:** Implementing the ability to show when a message has been read by the recipient.

- **Local Caching:** Utilizing IndexedDB or other local storage mechanisms for faster loading of message history.
- **Advanced Settings:** Adding user-configurable settings, such as file sending quality or key management.
- **File Compression:** Implementing file compression before sending to optimize bandwidth consumption and sending speed.

5.4. Conclusion:

This design document provides a robust technical foundation for developing the DeChat decentralized messenger based on the XMTP protocol. By adhering to this structure and utilizing modern tools like Vite, a stable, secure, and scalable product can be built. Success in this project requires precision in implementing technical details, as well as a deep understanding of the decentralized nature of the XMTP protocol and its features.