Docs

Project at a Glance

Elevator Pitch & Mission

Amadeus is a high-performance Layer 1 blockchain purpose-built to create, deploy, and monetize AI agents. It combines verifiable agent generation with no code required through Nova AI with real compute-mining via Useful Proof of Work, making it the first blockchain where AI agents evolve transparently, execute deterministically, and serve real users — all on-chain.

Why It Matters

Today's AI infrastructure is fragmented, centralized, and built for inference, not continuous learning or open deployment. Amadeus addresses this by:

- Letting anyone describe an agent idea in plain language, and have Nova AI compile it into verified, test-driven code.
- Enabling agents to live on-chain with memory, coordination logic, and real-time execution.
- Running all compute through a UPoW consensus model that re-purposes mining for real AI workloads.

Real-world example 1:

A founder needs a support agent that learns from product docs and handles on-boarding. With Amadeus, they describe the intent, and Nova builds and deploys the agent without any code. All logic and updates are verifiable, and usage fees flow back to the creator.

Real-world example 2:

A dev wants to run a decentralized research agent that scrapes, filters, and summarizes web data for traders. On Amadeus, this agent can evolve on-chain, maintain memory, and rely on a deterministic execution layer — without needing to set up infra or write smart contracts.

Key Links

- Website: https://ama.one
- X/Twitter: https://x.com/amadeusprotocol
- Discord: https://discord.gg/Cus2RwrhXB
- GitHub: https://github.com/amadeusprotocol
- Litepaper PDF: [link-to-litepaper.pdf] (upload when ready)

Disclaimer: The information on this documentation is provided for informational purposes only and does not constitute financial, investment, or legal advice. Amadeus is an experimental blockchain protocol under active development; features and use cases described are illustrative only and may change or never be implemented. $AMA is a utility token designed to pay for transaction fees and network operations. It is not intended as a security or investment, and no representation is made regarding profitability, appreciation, or future value. References to “monetization,” “fees,” or other economic outcomes are hypothetical examples only. Users are solely responsible for compliance with applicable laws, including those relating to securities, data protection, and intellectual property. Amadeus and its contributors disclaim all warranties and accept no liability for losses or damages arising from the use of the protocol or deployment of AI agents. Use at your own risk.

Ecosystem

Ecosystem: Builders, Validators, and Partners.

The Amadeus ecosystem consists of three core participant groups that collectively create, secure, and scale the network: Builders, Validators, and Partners. Each group contributes essential value to the growth of the $AMA token economy.

Builders create demand through agents and applications, Validators supply the secure compute that powers them, and Partners extend the ecosystem with storage, verification, data, and tooling.

Together, they form a unified value loop where builders rely on validators for compute and security, and validators verify the outputs and attestations enabled by Partner technologies, such as proofs and data feeds, which enhances reliability, verifiability, and performance across the network.

Builders

Builders are developers, founders, researchers, and creators building applications, agents, and infrastructure on top of the Amadeus network.

What Builders Do

- **Deploy smart contracts and dApps** on the WASM-based runtime
- **Create AI agents** using the Nova AI Compiler
- **Design agent swarms** for perception, planning, and reasoning tasks
- **Integrate data feeds** via AI Agent Oracle Streams
- **Build modules and extensions** for agent memory, identity, and verifiable compute
- **Participate in hackathons and grants** to prototype new primitives

How Builders Contribute to $AMA

- Consume $AMA for computation (agent execution fees)
- Pay gas fees for smart contract deployment
- Generate organic demand through usage of agents and dApps
- Onboard new users and external developers into the ecosystem
- Produce new primitives that attract partners and enterprise adoption

Validators

Validators secure the Amadeus network by participating in consensus and finalizing blocks through BLS-based validation. Instead of wasting energy on traditional hashing, Amadeus uses **Useful Proof of Work (uPoW)**, where validators perform real Tensorcore matrix computations that provide the decentralized compute powering AI workloads. This keeps the chain fast, reliable, and capable of supporting real-time agent activity.

In return, validators earn $AMA through block rewards and execution fees, making them essential to both network security and the long-term sustainability of the Amadeus economy.

Refer to the section Running a Node for technical setup of validator nodes and participation.

Partners

Partners in the Amadeus ecosystem are grouped by the capability layer they unlock. Each category strengthens a different part of the network and expands what builders and agents can do.

Infrastructure Partners

Provide the core technical building blocks for the network — such as data feeds, verifiable compute, confidential execution, and permanent storage.
Examples: oracles, ZK verification layers, TEEs, decentralized storage.

Liquidity Partners

Support the growth of the $AMA token economy through market making, exchanges, launchpads, and wallets that help users enter and move within the ecosystem.

Growth Partners

Bring new builders, users, and enterprises into Amadeus through hackathons, accelerators, venture studios, community programs, and co-marketing initiatives.

Arweave

Type: Infrastructure - Decentralized Data Storage
Website: https://arweave.org

Overview

Arweave is a decentralized, permanent storage network optimized for immutable data archiving and high-throughput access. **AR.IO'** s **Turbo bundling** solution efficiently handles billions of requests monthly, making it ideal for managing Amadeus AI agent datasets, model checkpoints, and provenance records at scale.

By integrating Arweave, Amadeus ensures **persistent, verifiable storage** for AI compute outputs without burdening the main chain — enabling a fast, secure, and decentralized layer for long-term agent memory and auditability.

Planned Capabilities for Amadeus

- Offload AI data (weights, logs, metadata) to permanent, verifiable Arweave storage.
- Use **Turbo bundling** via the TypeScript SDK or AR.IO API for high-throughput uploads and retrieval.
- Store lightweight **on-chain hashes** (e.g., SHA-256) for provenance without embedding full data in L1.
- Integrate **Rust Arweave crates** for native signing and uploads, ensuring authenticity and future Rust-Turbo compatibility.

Example Use Cases

AI Agent Data Persistence: Store and reference immutable model states, logs, and memory snapshots tied to agent IDs.

Compliance & Provenance: Enable audit trails for AI training datasets or model lineage through hashed metadata stored permanently.

Decentralized Knowledge Graphs: Archive structured outputs and public knowledge embeddings contributed by agent swarms.

Current Status

Integration planned using Rust-based Arweave crates for core signing and upload flows, combined with AR.IO 's Turbo TypeScript SDK for high-throughput data bundling.

Upcoming Deliverables (Planned)

Q1 2026: Implement hybrid integration—Rust crates for authenticated data uploads and Turbo SDK/API for high-throughput storage. Deploy a lightweight on-chain registry contract referencing stored data hashes for provenance and accounting.

Pontential extensions:

Extend support for multi-token payments (potentially including $AMA) and optimize Turbo bundling for agent-generated datasets and model checkpoints at scale.

zkVerify

Type: Infrastructure - Zero-Knowledge Compute Verification Layer

Website: https://zkverify.io/

Overview

What is zkVerify?

zkVerify is a zero-knowledge verification layer developed by Horizon Labs that enables proof-based validation of computational work — confirming correctness, authenticity, and ownership without exposing sensitive data.

By integrating zkVerify with Amadeus, the network gains verified compute capabilities where every training, inference, or agent action can be provably correct and privacy-preserving. This establishes Amadeus as a Layer 1 for verified compute, bridging decentralized AI workloads with enterprise-grade privacy and compliance.

Planned Capabilities for Amadeus

Once integrated, zkVerify will allow Amadeus to:

- Verify **uPoW (Useful Proof of Work)** jobs such as AI training or inference through zk-proofs.
- Enable **on-chain attestation** of AI computations with verifiable zk proofs validated via a zkVerify bridge.
- Provide **enterprise compliance features** such as provenance, geofencing, and licensing attestations.
- Establish a **proof settlement layer** between Amadeus and zkVerify for scalable, privacy-preserving AI execution.
- Support **zk verification for bridging** Amadeus with EVM-compatible chains using KZG-based proofs.

Example Use Cases

Verified AI Compute: AI training and inference workloads executed on Amadeus are verified through zk proofs, ensuring correctness and preventing manipulation.

Enterprise zk Compliance: Enterprises can deploy compliant AI applications with privacy-preserving proof attestations (provenance, data license, jurisdiction).

Cross-Chain zk Bridges: zkVerify-generated attestations validate Amadeus transactions for EVM-compatible networks, extending interoperability through verifiable proofs.

Current Status

Integration is under design and planned for rollout in two phases across Q4 2025–Q2 2026.

Coming Deliverables (Planned)

Phase 1: Q4 2025 – Q1 2026

Deploy zkVerify–Amadeus bridge and on-chain verifier contracts, enabling zk-proof validation for uPoW compute jobs and $AMA reward distribution.

Phase 2: Q2 2026

Scale to full AI computation verification with zkVerify as Amadeus' proof settlement layer, introducing enterprise-grade zk compliance for provenance, privacy, and licensing.

Crust Network

Type: Infrastructure - decentralized storage

Website: https://crust.network/

Crust Network is a decentralized, trustless storage layer built on IPFS and supported by thousands of globally distributed nodes. It provides censorship-resistant, verifiable storage for datasets, model artifacts, logs, and application state — all backed by a market-driven storage economy and TEE-enhanced security guarantees.

Integrated with Amadeus, Crust becomes the off-chain memory layer powering AI agents, oracle streams, model provenance, and long-term dataset storage. This positions Amadeus as a full-stack decentralized AI cloud: compute on Amadeus, storage on Crust.

Planned Capabilities for Amadeus

Once integration is complete, Crust Network will enable Amadeus to:

- **Store AI agent memory, logs, and checkpoints** in decentralized storage, with hashes anchored on-chain for provenance.
- **Extend Oracle Streams** by providing a durable archival layer for historical datasets, enrichments, and inference outputs.
- **Support persistent agent states** across long-running autonomous workflows with tamper-proof storage.
- **Enable decentralized dataset hosting**, allowing developers to provide training data that Amadeus agents can access via IPFS.
- **Provide cross-ecosystem wallet compatibility** (SubWallet, Talisman, Crust Wallet) for seamless user authentication and asset flow.
- **Create a durable data backend for uPoW compute**, keeping training artifacts and model updates accessible to verifiers, enterprises, and permissioned workflows.

Example Use Cases

Decentralized AI Memory

Amadeus agents persist memory, logs, and model checkpoints on Crust, ensuring tamper-proof lineage and long-term versioning.

Archival Layer for Oracle Streams

Live data served via Oracle Streams is continuously archived on Crust, enabling agents to learn from historical time-series datasets.

Dataset Hosting for AI Training

Developers, enterprises, or researchers can publish datasets to Crust for Amadeus agents to use during training or evaluation.

Cross-Chain User Access

Users from the Polkadot ecosystem can authenticate and interact with Amadeus using SubWallet or Talisman, enabling broader participation.

Durable Storage for uPoW Outputs

Training outputs and inference artifacts produced under Useful Proof of Work are stored securely and verifiably for downstream consumption.

Current Status

Integration is under active exploration, with technical design in progress and coordinated milestones across both teams.

Wallet

Creating a Wallet

Why You Need a Wallet

The Amadeus Wallet is your gateway to the Amadeus ecosystem. It serves as a secure digital vault that enables you to:

- **Store and manage AMA tokens**: Safely hold your AMA tokens and other supported cryptocurrencies
- **Send and receive transactions**: Transfer tokens to other users or receive payments
- **Connect to mining operations**: Link your wallet to mining software to receive mining rewards
- **Interact with the Amadeus blockchain**: Participate in the decentralized network
- **Track your portfolio**: Monitor your balance and transaction history in real-time

Without a wallet, you cannot:

- Receive mining rewards
- Participate in the Amadeus economy
- Store or transfer AMA tokens
- Access decentralized applications (dApps) on the Amadeus network

Supported Wallets

Currently, the Amadeus ecosystem supports:

Official Amadeus Web Wallet

- **Getting Started:** Web Wallet Tutorial
- **URL**: https://wallet.ama.one
- **Type**: Browser-based web wallet
- **Platforms**: Works on all modern browsers (Chrome, Firefox, Safari, Edge)
- **Features**:

    - No installation required
    - Cross-platform compatibility
    - Secure vault file system
    - Built-in transaction history
    - Real-time balance updates
    - Multiple token support

Official Amadeus Wallet Chrome Extension

- **Getting Started:** Chrome Extension Wallet Tutorial
- **Installation**: Available on Chrome Web Store
- **Type**: Chrome browser extension
- **Platforms**: Chrome and Chromium-based browsers
- **Features**:
    - Integrated browser experience
    - dApp connectivity
    - Transaction signing from web applications
    - Secure background service worker
    - Auto-lock functionality
    - Network configuration (Mainnet/Testnet/Custom)

Planned Support (Coming Soon)

- **Amadeus Mobile Wallet**: iOS and Android applications
- **Hardware Wallet Integration**: Support for hardware wallets
- **Third-party Wallet Support**: Integration with popular multi-chain wallets

Web Wallet

How to create a wallet

Creating a Wallet (Step-by-Step Guide)

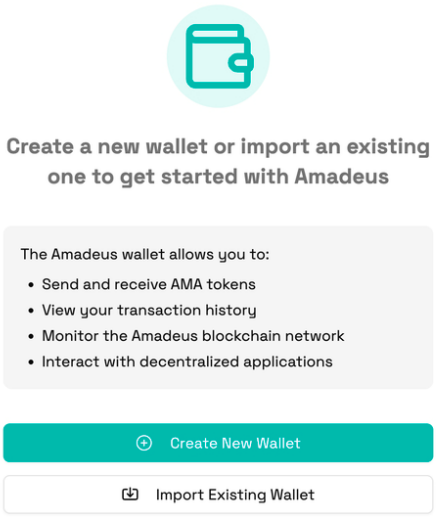
Creating an Amadeus wallet is a straightforward process that takes just a few minutes. Follow these steps carefully to ensure your wallet is set up securely.

Prerequisites

- Modern web browser (Chrome, Firefox, Safari, or Edge)
- Secure location to store your vault file
- Strong password that you can remember

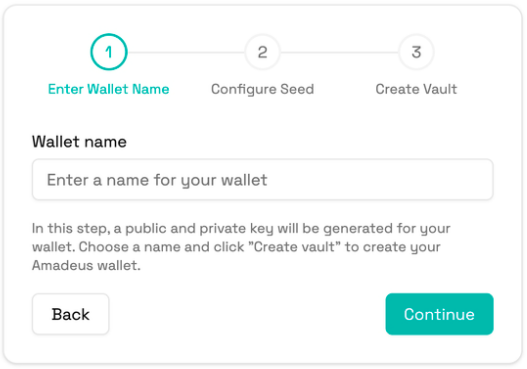
Step 1: Access the Wallet Application

1. Open your web browser
2. Navigate to https://wallet.ama.one
3. You'll see the wallet welcome screen with two options:
   - "Create New Wallet"
   - "Import Existing Wallet"



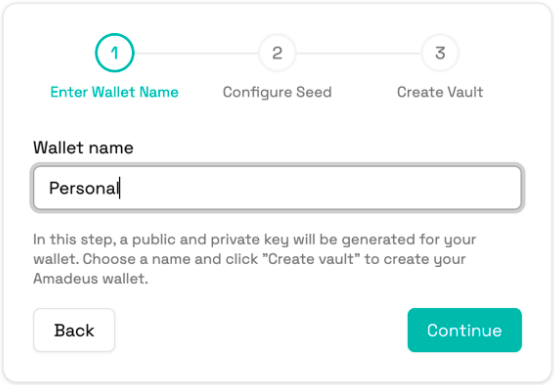
Step 2: Begin Wallet Creation

1. Click the **"Create New Wallet"** button
2. You'll be taken to the wallet creation wizard, which consists of three main steps:
   - Enter Wallet Name
   - Configure Seed
   - Create Vault



Step 3: Enter Wallet Name

1. **Wallet Name**: Enter a memorable name for your wallet
   - This is for your reference only and helps identify your wallet
   - Examples: "Main Wallet", "Mining Wallet", "Personal Wallet"
   - Maximum 50 characters allowed
   - Can contain letters, numbers, spaces, and special characters
2. Click **"Continue"** to proceed



Step 4: Configure Your Seed

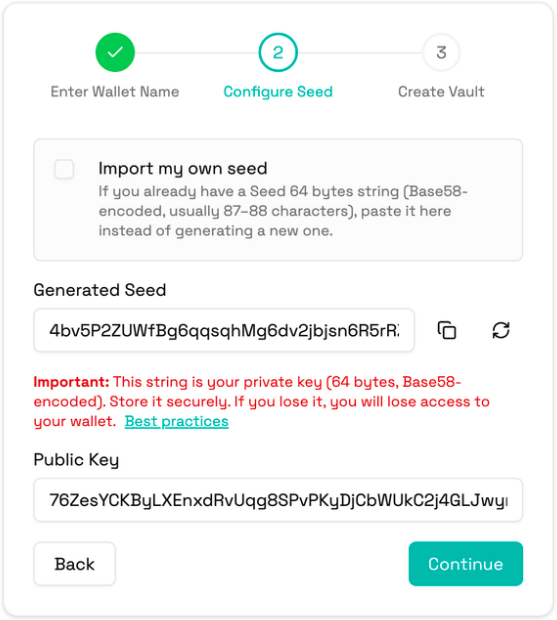
This is the most critical step in wallet creation. Your seed is your private key that controls access to your funds.

Understanding Seeds

- **Seed64**: A 64-byte (512-bit) cryptographic seed
- **Format**: Base58-encoded string (typically 87-88 characters)
- **Purpose**: Used to derive your wallet's public and private keys
- **Security**: This is your wallet's master key - never share it!

Automatic Seed Generation (Recommended)

1. By default, the wallet will automatically generate a secure random seed
2. Your generated seed will appear in the "Generated Seed" field
3. **Important**: Copy this seed immediately and store it securely
4. Click the **copy button** next to the seed field to copy it to your clipboard
5. The **refresh button** allows you to generate a new seed if desired



Using a Custom Seed (Advanced)

1. Toggle the **"Import existing seed"** checkbox

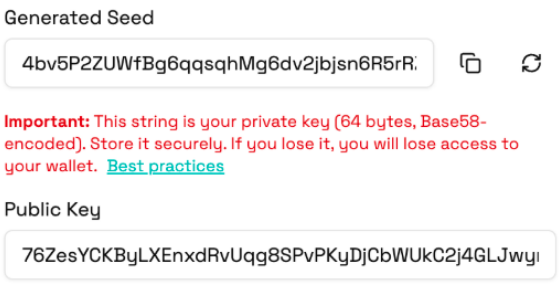
2. Paste your existing Seed64 into the field
3. The wallet will automatically derive and display your public key
4. This option is for:
    - Recovering an existing wallet
    - Importing a wallet from another device
    - Advanced users who generate seeds externally

Security Best Practices for Seeds

- **Never share your seed** with anyone
- **Store multiple copies** in secure locations
- **Consider using**:
    - Password manager
    - Encrypted USB drive
    - Paper backup in a safe
    - Safety deposit box
- **Never store** in:
    - Email
    - Cloud storage (unless encrypted)
    - Screenshots
    - Unencrypted text files

Step 5: Review Your Public Key

1. After seed configuration, your **public key** is automatically displayed
2. The public key is derived from your seed using BLS12-381 cryptography
3. This is your wallet address that others will use to send you tokens
4. Format: Base58-encoded string (typically 96 characters)

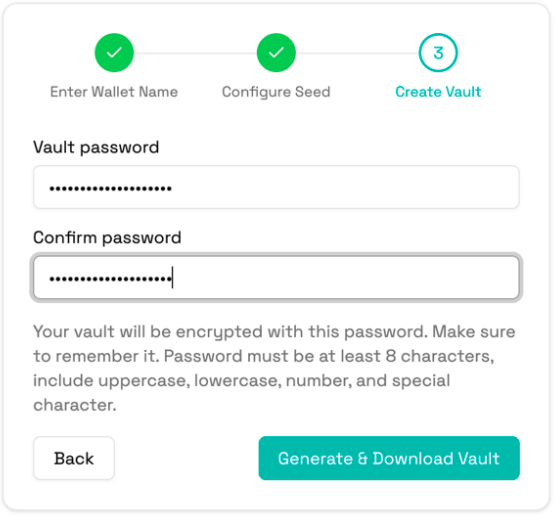


Step 6: Create Your Vault

The vault is an encrypted file that stores your wallet information securely.

Setting Your Vault Password

1. Enter a **strong password** for your vault
   - Minimum 8 characters
   - Must include:
     - Uppercase letter (A-Z)
     - Lowercase letter (a-z)
     - Number (0-9)
     - Special character (!@#$%^&*, etc.)
   - Example: "MySecure#Wallet2024!"
2. **Confirm your password** by entering it again



Understanding the Vault System

- **Encryption**: Uses AES-256-GCM encryption
- **Key Derivation**: PBKDF2 with 100,000 iterations
- **Salt**: Unique salt generated for each vault
- **Security**: Your seed is never stored in plain text

Step 7: Download Your Vault File

1. Click **"Create and Download Vault"**
2. Your browser will automatically download a JSON file
3. Default filename: `[WalletName]_vault.json`
4. **Critical**: Save this file in a secure location

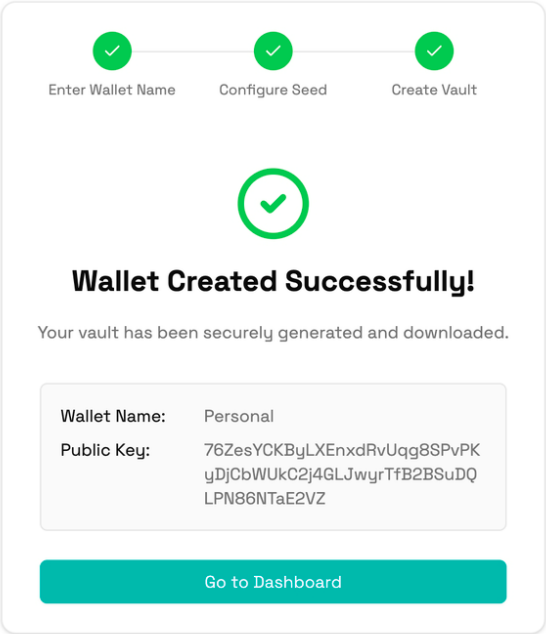
Vault File Contents

The vault file contains:

- Encrypted wallet data
- Wallet name
- Salt and initialization vector (IV)
- Creation timestamp
- NO plain text sensitive information

Step 8: Wallet Creation Success

1. You'll see a success screen confirming your wallet creation
2. Your wallet details are displayed:
   - Wallet name
   - Public key (address)
3. Click **"Go to Dashboard"** to access your wallet



Download and Install

Since the Amadeus Wallet is a web-based application, there's no traditional installation process. However, you can enhance your experience:

Accessing the Web Wallet

1. **Direct Access**: Visit https://wallet.ama.one
2. **Bookmark**: Add to your browser bookmarks for quick access
3. **Mobile**: Access from any mobile browser

Creating a Desktop Shortcut (Optional)

For Chrome/Edge:

1. Visit https://wallet.ama.one
2. Click the three-dot menu in the browser
3. Select "More tools" → "Create shortcut"
4. Check "Open as window" for an app-like experience
5. Click "Create"

For Firefox:

1. Visit https://wallet.ama.one
2. Resize the browser window to your preference
3. Drag the tab to your desktop
4. A shortcut will be created

Generating a New Wallet

Quick Generation Guide

For users who want to quickly generate a new wallet:

1. **Visit**: https://wallet.ama.one
2. **Click**: "Create New Wallet"
3. **Enter**: A wallet name
4. **Copy**: Your generated seed (CRITICAL - save this!)
5. **Create**: A strong vault password
6. **Download**: Your vault file
7. **Done**: Access your dashboard

Advanced Generation Options

Using External Seed Generators

Advanced users may generate seeds using:

- Hardware random number generators
- Cryptographic libraries
- Offline seed generation tools

Requirements for external seeds:

- Must be exactly 64 bytes (512 bits)
- Should use cryptographically secure randomness

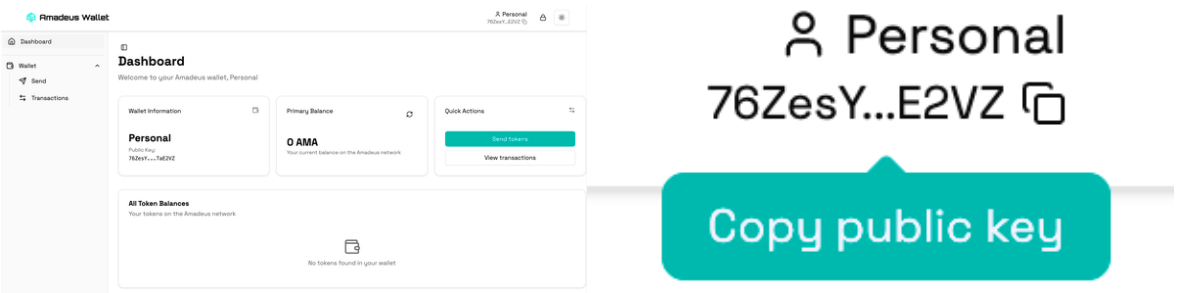
Receiving and Sending AMA Tokens

Receiving Tokens

Receiving AMA tokens is simple and secure:

Step 1: Access Your Wallet Address

1. Log into your wallet at https://wallet.ama.one
2. Navigate to the Dashboard
3. Your public key (wallet address) is displayed prominently
4. Click the **copy button** next to your address



Step 2: Share Your Address

Share your public key with the sender through:

- Secure messaging
- Email (address only, never seeds!)
- QR code (coming soon)
- In-person

Step 3: Confirm Receipt

1. Transactions typically confirm within seconds
2. Your balance updates automatically
3. View transaction details in the "Transactions" tab

Sending Tokens

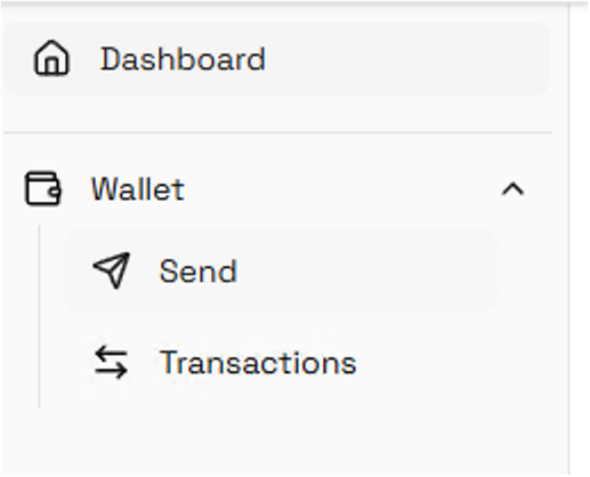
Sending AMA tokens requires an unlocked wallet:

Step 1: Unlock Your Wallet

1. If your session has expired, click "Unlock Wallet"
2. Enter your vault password
3. Your wallet unlocks for 30 minutes by default

Step 2: Navigate to Send

1. Click the **"Send"** button on the dashboard
2. Or navigate to Wallet → Send from the menu

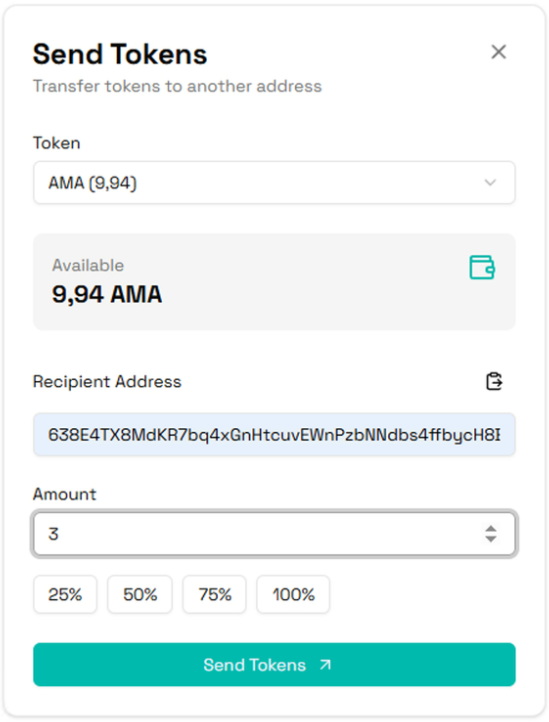


Step 3: Enter Transaction Details

1. **Recipient Address**:
   - Paste the recipient's Amadeus public key
   - Must be a valid Base58 address (48 bytes decoded)
   - Double-check the address - transactions are irreversible!
2. **Amount**:
   - Enter the amount to send
   - Can use percentage buttons (25%, 50%, 75%, 100%)
   - Displays available balance

- Supports decimal amounts

3. **Token Selection**:
   - Choose which token to send (default: AMA)
   - Shows balance for selected token
   - Additional tokens supported as ecosystem grows



Step 4: Confirm and Send

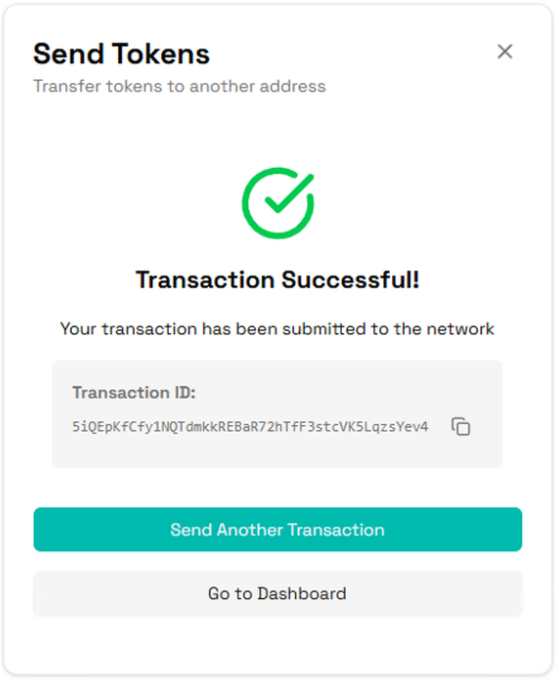
1. Review transaction details carefully:
   - Recipient address (verify first/last characters)
   - Amount and token type
   - Current balance
2. Click **"Send Tokens"**
3. Transaction is signed locally with your private key
4. Submitted to the Amadeus network

Step 5: Transaction Confirmation

1. Success screen shows:
   - Transaction ID (hash)

- Confirmation status
- Option to send another transaction

2. Transaction typically confirms in:
   - 1-3 seconds for standard transactions
   - Up to 10 seconds during high network activity



Transaction Fees

TBD: Fee structure may be implemented in future network upgrades

Connecting Your Wallet to Mining

To receive mining rewards, you need to import the mining wallet that was automatically created when you started mining.

Prerequisites

- Amadeus mining software already installed and configured
- Access to your home directory where mining keys are stored
- The wallet creation steps completed from previous sections

Important: Mining Creates Its Own Wallet

When you first run the Amadeus miner, it automatically generates a private key and stores it in your system. To access your mining rewards, you must import this key into your web wallet.

Step-by-Step: Import Your Mining Wallet

Step 1: Locate Your Mining Private Key

1. **Open a terminal** on your mining machine
2. **Navigate to the Amadeus cache directory**:

```
cd ~/.cache/amadeus/
```

3. **Display your private key**:

```
more sk
```

4. **Copy the displayed key** - this is your mining wallet's private key (Seed64)

[Placeholder: Screenshot of terminal showing sk file location]

Step 2: Import to Web Wallet

1. **Go to** https://wallet.ama.one
2. **Click** "Create New Wallet"
3. **Enter** a wallet name (e.g., "Mining Wallet")
4. **Check** the "Import existing seed" checkbox
5. **Paste** the private key from the `sk` file
6. **Continue** with vault creation as described in previous sections

```
cd ~/.cache/amadeus/
ls -ltr
total 8
-rw-rw-r-- 1 amadeus amadeus   88 Jun  5 00:16 sk
drwxr-xr-x 3 amadeus amadeus 4096 Jun 23 17:01 db

more sk
<your secret key will be displayed>
```

Security Considerations

⚠️ **Critical Security Notes**:

- The `sk` file contains your mining wallet's private key
- **Back up this file** immediately to prevent loss of mining rewards
- **Never share** the contents of this file
- **Store securely** using the same practices as your main wallet seed

Backing Up Your Mining Key

```
# Create a secure backup
cp ~/.cache/amadeus/sk ~/mining_wallet_backup_$(date +%Y%m%d).txt

# Set appropriate permissions
chmod 600 ~/mining_wallet_backup_*.txt
```

Verifying Your Mining Wallet

1. **After importing**, check your wallet dashboard
2. **Your balance** will show accumulated mining rewards
3. **Transaction history** will display all mining payouts
4. **Public key** shown in wallet matches your miner's address

Alternative: Using Multiple Wallets

You can also:

1. **Keep mining wallet separate** for security

2. **Transfer funds** periodically to your main wallet
3. **Use mining wallet** exclusively for mining operations

Troubleshooting

Cannot Find sk File

- Ensure miner has run at least once
- Check correct path: `~/.cache/amadeus/`
- Verify permissions to access the directory

Import Fails

- Confirm the key is exactly as shown in the file
- No extra spaces or line breaks
- Must be valid Base58 format

No Mining Rewards Showing

- Mining rewards appear after blocks are mined
- Check the mining documentation for payout schedules
- Ensure miner is properly configured and running

Next Steps

After successfully importing your mining wallet:

1. **Monitor your mining rewards** on the dashboard
2. **Set up regular backups** of your mining keys
3. **Consider transferring** rewards to a main wallet periodically
4. **Review security practices** for both wallets

Security Best Practices

Essential Security Measures

1. Seed Protection

- **Never share** your seed with anyone
- **Never enter** your seed on suspicious websites
- **Store offline** in multiple secure locations
- **Use encryption** for digital backups
- **Consider** splitting seed across locations (advanced)

2. Vault File Security

- **Encrypt** vault file before cloud storage
- **Use unique** passwords for each vault
- **Store separately** from seed backups
- **Regular backups** to prevent loss
- **Version control** for vault updates

3. Password Management

- **Use strong**, unique passwords for vaults
- **Enable** password manager integration
- **Never reuse** passwords across services
- **Change passwords** if compromise suspected

4. Device Security

- **Keep browsers** updated
- **Use antivirus** software
- **Avoid public** WiFi for transactions
- **Clear cache** after wallet sessions
- **Use dedicated** device for large holdings

Operational Security (OpSec)

Public Information

Safe to share:

- ✅ Your public key (wallet address)
- ✅ Transaction IDs
- ✅ General balance ranges (if comfortable)

Never share:

- ❌ Your seed/private key
- ❌ Vault passwords
- ❌ Exact holdings
- ❌ Personal identifying information

Phishing Protection

- **Bookmark** https://wallet.ama.one
- **Verify URL** before entering passwords
- **Check SSL** certificate (padlock icon)
- **Ignore emails** asking for seeds
- **Report phishing** to community moderators

Physical Security

- **Secure workstation** when away
- **Lock screens** during breaks
- **Private location** for wallet access
- **Shoulder surfing** awareness
- **Clean desk** policy for seed storage

Recovery Planning

Backup Strategy

1. **3-2-1 Rule**:
    - 3 copies of important data

- 2 different storage media
- 1 offsite backup

2. **Test Recovery**:
   - Regularly verify backup integrity
   - Practice recovery procedures
   - Document recovery steps
   - Update backups after changes

Emergency Access

- **Trusted contacts** for emergency (lawyer, family)
- **Clear instructions** for recovery
- **Legal documentation** for inheritance
- **Time-locked** recovery options (planned feature)

FAQs (Wallet Section)

General Questions

Q: Is the Amadeus Wallet free to use? A: Yes, the Amadeus Wallet is completely free. There are no charges for creating wallets, storing tokens, or accessing features.

Q: Can I create multiple wallets? A: Yes, you can create as many wallets as needed. Each wallet requires its own seed and vault file.

Q: What browsers are supported? A: Modern versions of Chrome, Firefox, Safari, and Edge. Mobile browsers are also supported.

Security Questions

Q: Where is my private key stored? A: Your private key is encrypted within your vault file and only decrypted in-memory when you unlock your wallet. It's never sent to servers or stored in plain text.

Q: What happens if I lose my vault password? A: If you have your seed backed up, you can create a new wallet by importing your seed. Without the seed, funds cannot be recovered.

Q: Is my wallet data sent to Amadeus servers? A: No. The wallet operates client-side. Only public blockchain queries (balance, transactions) communicate with Amadeus nodes.

Q: Can Amadeus team access my wallet? A: No. The wallet is non-custodial. Only you have access to your seeds and private keys.

Technical Questions

Q: What cryptography does the wallet use? A:

- **Key Generation**: BLS12-381 elliptic curve
- **Encryption**: AES-256-GCM
- **Key Derivation**: PBKDF2 with SHA-256
- **Hashing**: BLAKE3 for transaction hashes

Q: What is the wallet session timeout? A: Default timeout is 30 minutes. The session extends with activity. After timeout, you'll need to re-enter your vault password.

Q: Can I adjust transaction fees? A: Currently, Amadeus has no transaction fees. This may change with future network updates.

Q: What's the difference between seed and private key? A: Your seed is 64 bytes and used to derive your private key (32 bytes) through cryptographic reduction. The seed is your master secret.

Troubleshooting

Q: My balance isn't updating A:

1. Check your internet connection
2. Refresh the page
3. Click the refresh button on the balance card

4. Verify the transaction was sent to the correct address

Q: I can't unlock my wallet A:

1. Ensure you're using the correct vault file
2. Verify your password (check caps lock)
3. Try re-uploading the vault file
4. Confirm the vault file isn't corrupted

Q: Transaction failed A:

1. Check recipient address format
2. Verify sufficient balance
3. Ensure wallet is unlocked
4. Try again after a few moments
5. Check network status

Q: Can't connect wallet to mining A:

1. Use your public key, not seed
2. Remove any extra spaces in address
3. Restart mining software after configuration
4. Verify pool accepts Amadeus addresses

Future Features

Q: What features are planned? A: Upcoming features include:

- Mobile applications (iOS/Android)
- Hardware wallet support
- Multi-signature wallets
- QR code transactions
- Advanced portfolio analytics
- DeFi integrations
- Plugin wallet

Chrome Extension Wallet

Welcome to the Amadeus Wallet Extension documentation. This guide will help you get started with the wallet and understand all its features.

Table of Contents

Overview

The Amadeus Wallet Extension is a secure Chrome extension wallet for the Amadeus blockchain. It provides a user-friendly interface for managing your AMA tokens and interacting with decentralized applications (dApps).

Key Features

- **Secure Wallet Management**: Create or import wallets with encrypted storage
- **Token Transfers**: Send and receive AMA tokens easily
- **Transaction History**: View all your past transactions
- **dApp Integration**: Sign transactions from web applications
- **Network Configuration**: Switch between Mainnet and Testnet
- **Security Controls**: Auto-lock, Seed64 backup, and more

Security Model

- Private keys are never stored in plain text
- All sensitive data is encrypted using AES-GCM
- Automatic wallet locking after inactivity
- Private keys remain in the background service worker (never in UI state)

1. Getting Started

This guide will help you install and set up the Amadeus Wallet Extension.

Installation

Prerequisites

- Google Chrome browser (or Chromium-based browser)
- Internet connection

Installation Steps

1. **Install from Chrome Web Store**
   - Visit the Amadeus Wallet page on Chrome Web Store
   - Click the **"Add to Chrome"** button
   - Review the permissions requested by the extension
   - Click **"Add Extension"** to confirm installation
2. **Verify Installation**
   - The extension will be installed automatically
   - You'll see a confirmation message
   - The Amadeus Wallet icon should appear in your Chrome toolbar
3. **Pin the Extension** (Recommended)
   - Click the puzzle icon (extensions menu) in Chrome's toolbar
   - Find "Amadeus Wallet" in the list
   - Click the pin icon next to it
   - The wallet icon will now appear directly in your toolbar for easy access

First Launch

When you first open the Amadeus Wallet Extension, you'll see the onboarding screen with two options:

- **Create New Wallet**: Generate a new wallet with a new Seed64

- **Import Existing Wallet**: Restore a wallet using your Seed64

Choose the option that applies to you and follow the respective guides:

- Creating a Wallet
- Importing a Wallet

Extension Icon

The Amadeus Wallet icon appears in your browser toolbar. Click it to:

- View your wallet balance
- Access all wallet features
- Manage your account settings

Next Steps

After setting up your wallet, you can:

- Unlock your wallet to start using it
- Send tokens to other addresses
- Receive tokens by sharing your address
- View transaction history

2. Creating a New Wallet

This guide will walk you through creating a new Amadeus wallet.

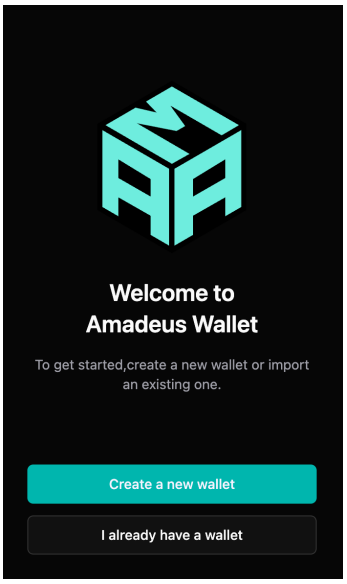
Overview

Creating a new wallet generates a unique Seed64 (64-byte cryptographic seed) that you must securely store. This Seed64 is the only way to recover your wallet if you lose access to your device.

Step-by-Step Guide

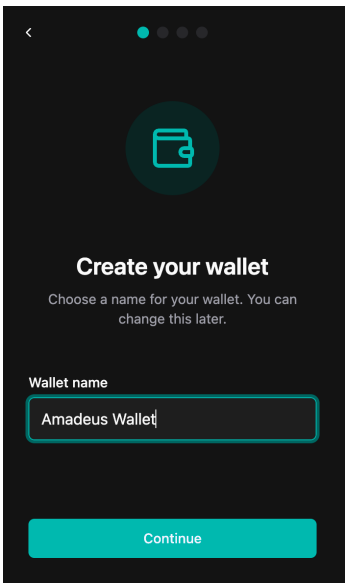
Step 1: Choose "Create New Wallet"

When you first open the extension, click **"Create New Wallet"** on the onboarding screen.



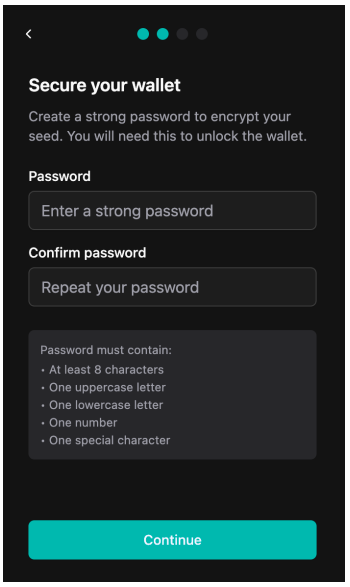
Step 2: Enter Wallet Name

1. Enter a name for your wallet (e.g., "My Main Wallet")
2. This name helps you identify the wallet and is stored locally
3. Click **"Continue"**



Step 3: Set Password

1. Create a strong password for your wallet
   - Minimum requirements: At least 8 characters
   - Use a combination of letters, numbers, and special characters
   - Make it unique and memorable
2. Confirm your password by entering it again
3. Click **"Continue"**



> ⚠️ **Important**: Remember this password! You'll need it every time you unlock your wallet. If you forget it, you'll need your Seed64 to reset it.

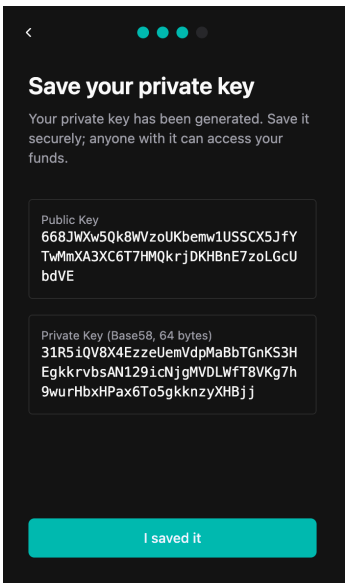
Step 4: Save Your Seed64

This is the **most critical step**. Your Seed64 is a 64-byte (512-bit) cryptographic seed that allows you to restore your wallet.

Understanding Seed64:

- **Seed64**: A 64-byte (512-bit) cryptographic seed used to derive your wallet's keys
- **Format**: Base58-encoded string (typically 87-88 characters)
- **Purpose**: Used to derive your wallet's public and private keys
- **Security**: This is your wallet's master key - never share it!

1. **Copy your Seed64**
   - The extension will display your Seed64
   - Click the copy button to copy it to your clipboard
   - The Seed64 is a long string of characters (Base58 encoded)
2. **Store it securely**
   - Write it down on paper or store in encrypted format
   - Keep it in a safe place (fireproof safe, bank vault)
   - Never share it with anyone
   - Never store it digitally in unencrypted form (screenshots, cloud storage, email, etc.)
   - Consider making multiple copies stored in different secure locations
   - Consider using password manager (encrypted) or encrypted USB drive
3. **Verify you've saved it correctly**
   - Double-check that you've copied the complete Seed64
   - Ensure no characters are missing
   - Click the checkbox to confirm you've saved it
   - Click **"Continue"**

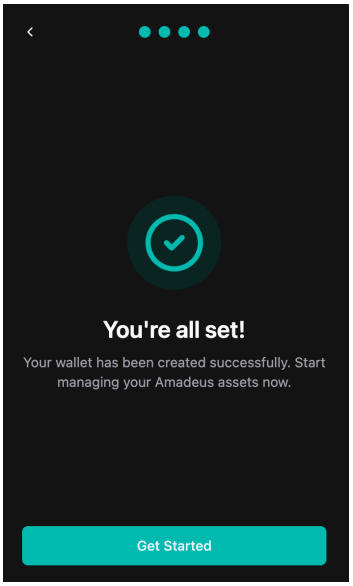


> ⚠️ **Critical**: Your Seed64 is the only way to recover your wallet. Store it securely and never share it!

Step 6: Wallet Created

Congratulations! Your wallet has been created successfully.

- Your wallet address (public key) will be displayed
- You can copy it to share with others for receiving tokens
- Click **"Go to Dashboard"** to start using your wallet



Security Best Practices

Seed64 Protection

- ✅ Write Seed64 on paper or store encrypted
- ✅ Store in multiple secure locations (3-2-1 backup rule: 3 copies, 2 media types, 1 offsite)
- ✅ Use encryption for digital backups
- ✅ Consider splitting Seed64 across locations (advanced)
- ❌ Never share your Seed64 with anyone
- ❌ Never store Seed64 digitally in unencrypted form (screenshots, cloud storage, email)
- ❌ Never enter Seed64 on suspicious websites
- ❌ Never take screenshots of your Seed64

Password Security

- ✅ Use a strong, unique password (minimum 8 characters)
- ✅ Include uppercase, lowercase, numbers, and special characters
- ✅ Use a password manager for your wallet password
- ✅ Never reuse passwords across services

Troubleshooting

I forgot my password

- You can reset your password using your Seed64 through the "Forgot Password" option

I lost my Seed64

- Unfortunately, without your Seed64, you cannot recover your wallet
- This is why it's critical to store it securely

I want to create another wallet

- You can create multiple wallets, but each requires a separate Seed64
- Consider using different wallets for different purposes (main, savings, testing)

What's Next?

- Unlock your wallet to access your dashboard
- Receive tokens by sharing your wallet address
- Send tokens once you have tokens in your wallet

3. Importing an Existing Wallet

Learn how to import an existing Amadeus wallet using your Seed64.

Overview

If you already have a wallet with a Seed64, you can import it into the extension to access your funds and transaction history.

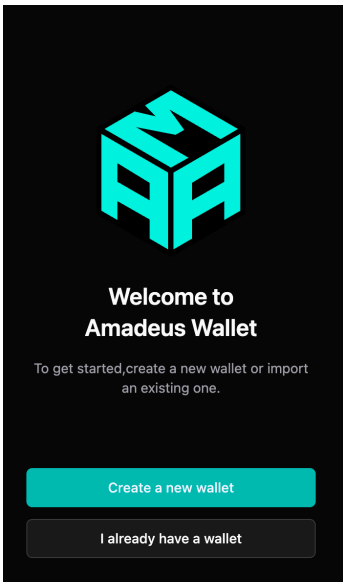
Prerequisites

- Your Seed64 (64-byte seed, Base58-encoded string)
- Access to the wallet extension

Step-by-Step Guide

Step 1: Choose "I already have a wallet"

When you first open the extension, click **"Import Existing Wallet"** on the onboarding screen.

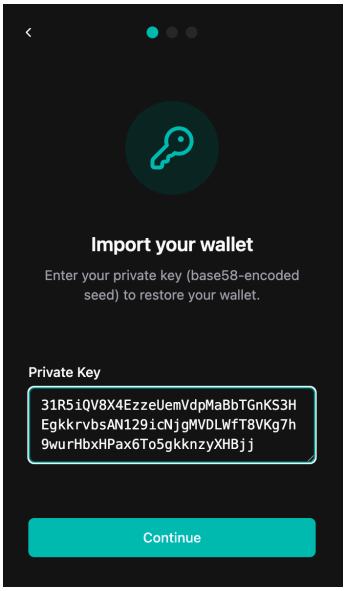


Step 2: Enter Your Seed64

1. Enter your Seed64

- Paste your Seed64 string (Base58-encoded, typically 87-88 characters)
- The extension will validate the format as you type
- Ensure you have the complete Seed64 string

2. **Double-check your entry**
   - Verify all characters are correct
   - Ensure no characters are missing
   - Check for any extra spaces or line breaks
   - The Seed64 should be a single continuous string
3. Click **"Continue"**

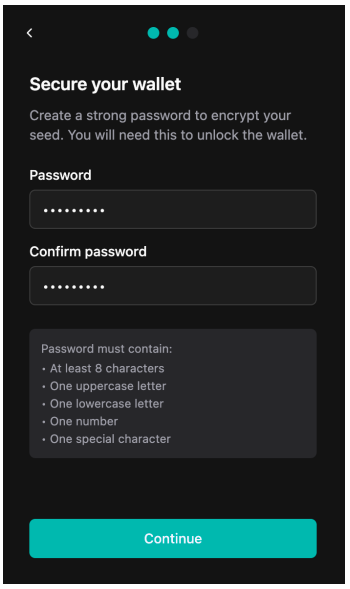


> ⚠️ **Security Note**: Make sure you're entering your Seed64 in a private, secure location. Never share your Seed64 with anyone.

Step 3: Set Password

1. Create a password for this wallet instance
   - This password protects the wallet on this device
   - It can be different from passwords used on other devices
   - Minimum 8 characters recommended
2. Confirm your password
   - Re-enter the password to confirm

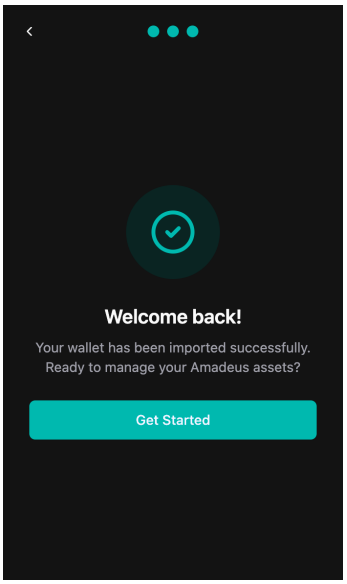
- Click **"Continue"**



Step 4: Wallet Imported Successfully

Your wallet has been successfully imported!

- Your wallet address will be displayed
- Your balance and transaction history will sync
- Click **"Get Started"** to access your wallet



Verification

After importing, verify that:

- ✅ Your wallet address matches your expected address
- ✅ Your balance is correct (may take a moment to sync)
- ✅ Your transaction history appears (if any)

Troubleshooting

"Invalid Seed64" error

Possible causes:

- Incorrect Seed64 format
- Missing characters
- Extra spaces or line breaks
- Invalid Base58 encoding

Solutions:

- Double-check your Seed64 against your original backup
- Ensure the complete Seed64 string is entered
- Remove any extra spaces or line breaks
- Verify you're using the correct Seed64 for this wallet
- Ensure it's a valid Base58-encoded string (87-88 characters)

Balance not showing

Possible causes:

- Network connection issues
- Wrong network selected (Mainnet vs Testnet)
- Transaction history still syncing

Solutions:

- Check your internet connection
- Verify network settings in Settings → Networks
- Wait a few moments for sync to complete
- Refresh the extension

Transaction history missing

- Transaction history may take a few moments to load
- Ensure you're on the correct network (Mainnet/Testnet)
- Check that the wallet address matches your expected address

Security Reminders

- ✅ Only import wallets on trusted devices
- ✅ Ensure your device is secure and malware-free
- ✅ Never share your Seed64
- ✅ Use a strong password for the wallet
- ✅ Consider using a hardware wallet for large amounts

What's Next?

- Unlock your wallet to access your dashboard
- View your transaction history
- Send tokens if you have a balance
- Configure network settings if needed

4. Unlocking Your Wallet

Learn how to unlock your Amadeus wallet to access your funds and perform transactions.

Overview

Your wallet automatically locks after a period of inactivity for security. You'll need to unlock it with your password to access your funds and perform transactions.

When You Need to Unlock

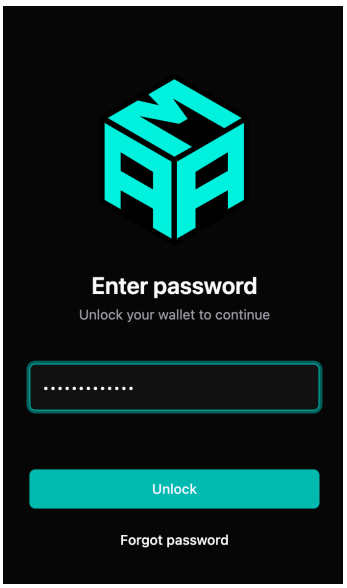
You'll be prompted to unlock your wallet when:

- You first open the extension after installation
- The wallet has been locked due to inactivity
- You manually locked the wallet from Settings
- You restart your browser

Unlocking Process

Step 1: Enter Password

1. Open the Amadeus Wallet Extension
2. Enter your wallet password
3. Click **"Unlock"**

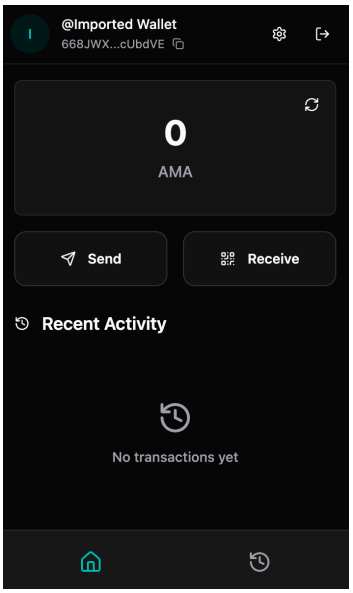


> 💡 **Tip**: If you've forgotten your password, you can reset it using your Seed64 via "Forgot Password".

Step 2: Access Granted

Once unlocked, you'll be taken to your dashboard where you can:

- View your balance
- Send and receive tokens
- View transaction history
- Access all wallet features



Auto-Lock Feature

The wallet automatically locks after a period of inactivity to protect your funds. The default auto-lock duration is **15 minutes** (configurable up to 30 minutes or more).

Configuring Auto-Lock

You can change the auto-lock duration in Settings:

1. Go to **Settings** → **Security & Privacy**
2. Find **"Auto-lock Duration"**
3. Select your preferred duration:
   - 5 minutes
   - 15 minutes (default)
   - 30 minutes
   - 1 hour
   - Never (not recommended)

> ⚠️ **Security Warning**: Setting auto-lock to "Never" means your wallet stays unlocked until you manually lock it. This is less secure and not recommended.

Manual Locking

You can manually lock your wallet at any time:

1. Go to **Settings**
2. Click **"Lock Wallet"** button at the bottom
3. Your wallet will be locked immediately

Alternatively, you can lock from the dashboard:

- Click the lock icon in the header
- Or use the lock button in Settings

Forgot Password

If you've forgotten your password:

1. Click **"Forgot Password"** on the unlock screen
2. Enter your Seed64 (Base58-encoded string)
3. Create a new password
4. Confirm the new password
5. Your wallet will be unlocked with the new password

> ⚠️ **Important**: You must have your Seed64 to reset your password. Without it, you cannot access your wallet.

Security Best Practices

- ✅ Use a strong, unique password
- ✅ Enable auto-lock for security
- ✅ Lock your wallet when leaving your device unattended
- ✅ Never share your password
- ✅ Consider using a password manager for your wallet password
- ❌ Don't disable auto-lock
- ❌ Don't write your password in plain text

Session Management

When your wallet is unlocked:

- Your session remains active until auto-lock triggers
- Performing actions extends your session
- Closing the extension popup doesn't lock the wallet
- The wallet stays unlocked in the background until auto-lock

Troubleshooting

"Incorrect password" error

Possible causes:

- Typo in password
- Caps Lock enabled
- Wrong password for this wallet

Solutions:

- Double-check your password entry
- Verify Caps Lock is off
- Try typing the password slowly
- Use "Forgot Password" if needed

Wallet won't unlock

- Ensure you're using the correct password
- Check that your browser extension is up to date
- Try refreshing the extension page
- If all else fails, use "Forgot Password" with your Seed64

What's Next?

Once unlocked, you can:

- Send tokens
- Receive tokens
- View transaction history
- Sign transactions from dApps

5. Sending Tokens

Learn how to send AMA tokens to other addresses using the Amadeus Wallet Extension.

Overview

Sending tokens allows you to transfer AMA tokens from your wallet to another address on the Amadeus blockchain.

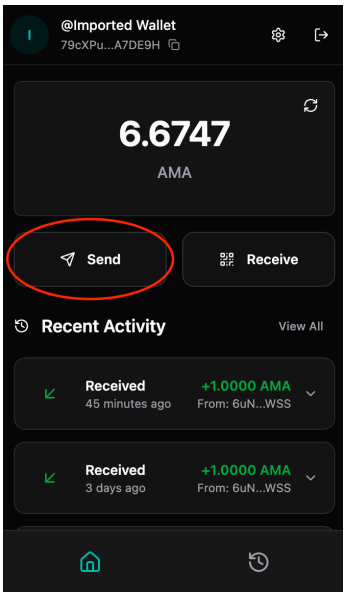
Prerequisites

- Wallet must be unlocked
- Sufficient balance in your wallet
- Recipient's wallet address

Step-by-Step Guide

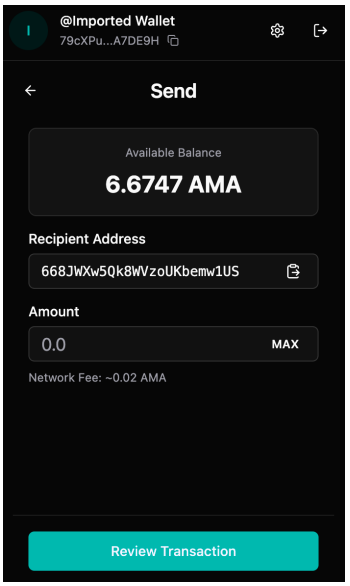
Step 1: Access Send Page

1. Open the Amadeus Wallet Extension
2. Ensure your wallet is unlocked
3. Click **"Send"** button on your dashboard



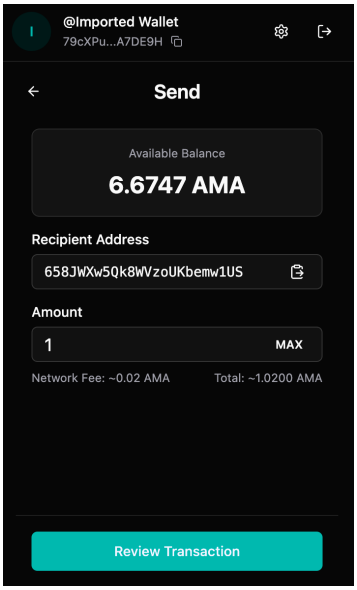
Step 2: Enter Recipient Address

1. In the **"To"** field, enter the recipient's wallet address
   - You can paste the address directly
   - The extension validates the address format
   - Invalid addresses will show an error
2. **Verify the address** before proceeding
   - Double-check each character
   - Ensure it's the correct address (transactions cannot be reversed)
   - Consider copying and pasting to avoid typos



Step 3: Enter Amount

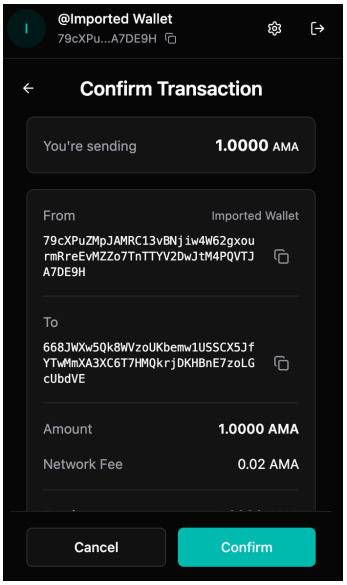
1. In the **"Amount"** field, enter the amount you want to send
   - You can enter a decimal number (e.g., 100.5)
   - The amount is in AMA tokens
   - Your available balance is shown below the field
2. **Check your balance**
   - Ensure you have enough tokens
   - Remember: Transaction fees may apply
   - The extension will warn you if balance is insufficient
3.



Step 4: Review Transaction

Before confirming, review:

- ✅ Recipient address (correct and complete)
- ✅ Amount (correct value)
- ✅ Token type (correct token)
- ✅ Your balance (sufficient funds)
- ✅ Network (Mainnet or Testnet)



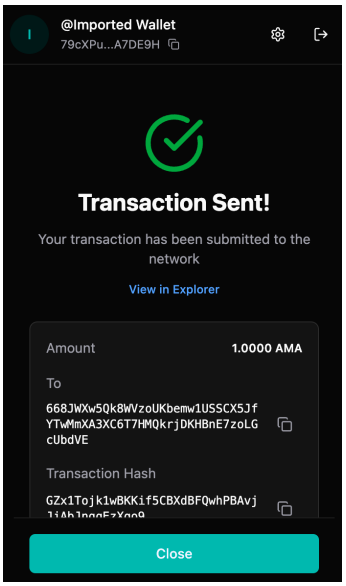
Step 5: Confirm Transaction

1. Click **"Confirm"**
2. The extension will process your transaction
3. You'll see a "Processing" screen while the transaction is being submitted

Step 6: Transaction Complete

Once processed, you'll see:

- ✅ Transaction hash (unique identifier)
- ✅ Success message
- ✅ Link to view transaction in explorer
- ✅ Option to return to dashboard



Transaction Details

Transaction Hash

Every transaction has a unique hash (transaction ID) that:

- Identifies your transaction on the blockchain
- Can be used to track transaction status
- Can be shared with the recipient for verification
- Can be viewed in the blockchain explorer

Viewing in Explorer

Click **"View in Explorer"** to:

- See transaction details on the blockchain
- Verify transaction status
- Check confirmation status
- View transaction on the public ledger

Transaction Fees

- Transaction fees are automatically calculated
- Fees are deducted from your balance
- Fee amount depends on network congestion
- Fees are typically minimal

Transaction Status

Your transaction can have different statuses:

- **Pending**: Transaction submitted, waiting for confirmation
- **Confirmed**: Transaction included in a block
- **Failed**: Transaction failed (insufficient funds, invalid address, etc.)

Security Tips

- ✅ Always verify the recipient address
- ✅ Double-check the amount before sending
- ✅ Start with a small test transaction for new addresses
- ✅ Verify you're on the correct network (Mainnet vs Testnet)
- ❌ Never send to addresses you don't trust
- ❌ Be cautious of phishing attempts asking for tokens

Troubleshooting

"Insufficient balance" error

Causes:

- Not enough tokens in your wallet
- Transaction fees exceed available balance
- Network fees increased

Solutions:

- Check your current balance
- Reduce the amount you're sending
- Ensure you have enough for fees

Transaction stuck/pending

Possible causes:

- Network congestion
- Low transaction fee
- Network issues

Solutions:

- Wait for network confirmation
- Check transaction status in explorer
- Verify network connectivity
- Contact support if stuck for extended period

Wrong address entered

Important: Transactions cannot be reversed!

If you haven't confirmed yet:

- Simply cancel and re-enter the correct address

If already sent:

- Unfortunately, tokens sent to wrong addresses cannot be recovered
- Always double-check addresses before confirming

Transaction failed

Common causes:

- Invalid recipient address
- Insufficient balance
- Network error
- Contract execution error

Solutions:

- Verify recipient address format
- Check your balance
- Try again after checking network status
- Review error message for specific issue

Best Practices

1. **Test First**: Send a small amount first when using a new address
2. **Verify Address**: Always double-check recipient addresses
3. **Check Network**: Ensure you're on the correct network
4. **Keep Records**: Save transaction hashes for your records
5. **Monitor Balance**: Keep track of your balance after transactions

What's Next?

- View your transaction history
- Receive tokens from others

- Configure network settings if needed

6. Receiving Tokens

Learn how to receive AMA tokens in your Amadeus wallet.

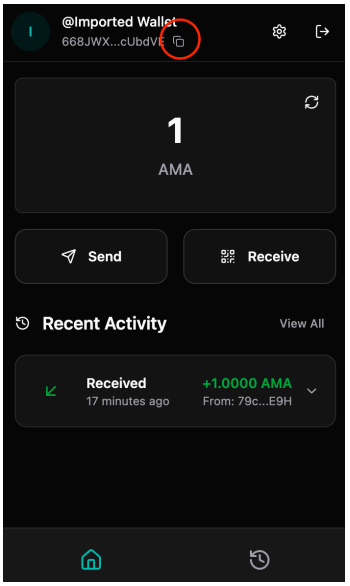
Overview

Receiving tokens is simple - you just need to share your wallet address with the sender. This guide explains how to find and share your address.

Finding Your Wallet Address

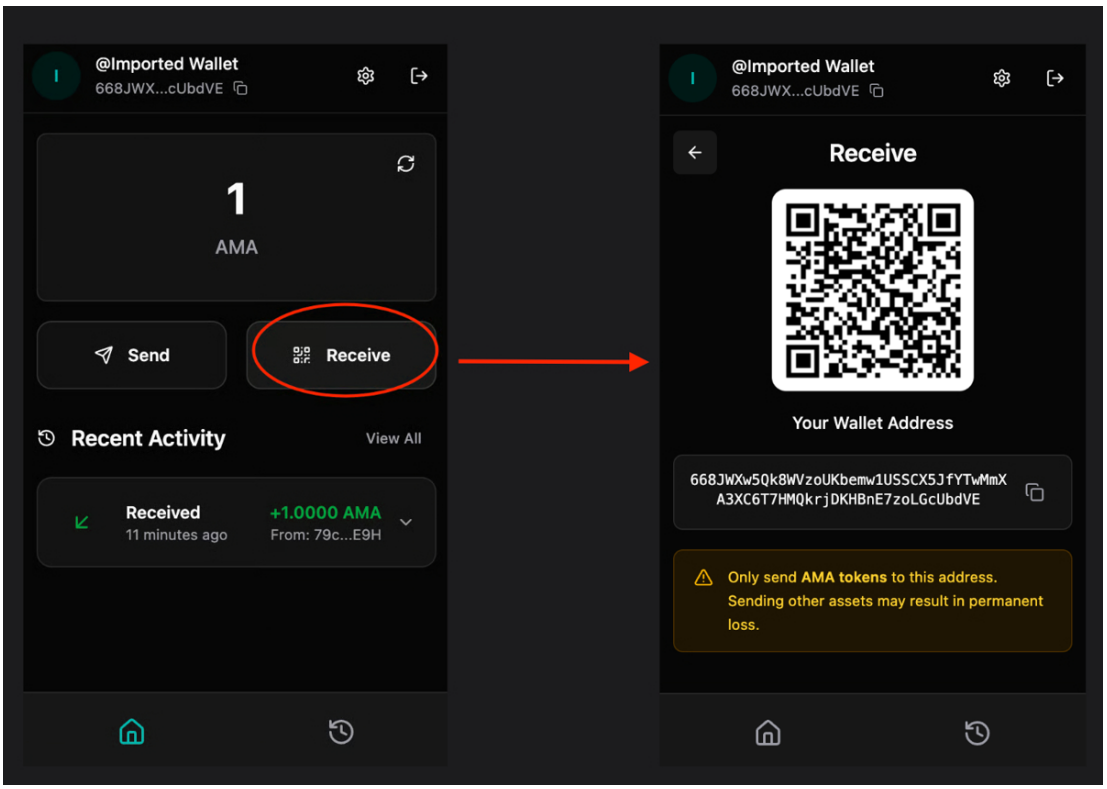
Method 1: From Header

1. Open the Amadeus Wallet Extension
2. Unlock your wallet if needed
3. Your wallet address is displayed on the header
4. Click the **copy icon** next to your address to copy it



Method 2: From Receive Page

1. Click **"Receive"** in the navigation menu
2. Your wallet address and QR code will be displayed
3. Click **"Copy Address"** to copy your address



Sharing Your Address

Option 1: Copy and Paste

1. Copy your wallet address (using the copy button)
2. Share it via:
   - Email
   - Messaging apps
   - Social media
   - Any communication method

Option 2: QR Code

1. Go to the **Receive** page
2. Show the QR code to the sender
3. They can scan it with their wallet app
4. This ensures accuracy and prevents typos

Option 3: Share QR Code Image

1. Take a screenshot of your QR code
2. Send the image to the sender
3. They can scan it from the image

Address Format

Your wallet address:

- Is a long string of characters (Base58 encoded)
- Starts with specific characters depending on the network
- Is unique to your wallet
- Can be shared publicly (it's your public key)

Example format:

```
5KJvsngHeMoo884xkJ6Cyb5StvnRN6f9tYiqwqJzLpQq
```

QR Code

The QR code contains:

- Your wallet address
- Can be scanned by other wallet apps
- Ensures accurate address entry
- Faster than manual entry

Receiving Tokens

Step 1: Share Your Address

Share your wallet address or QR code with the sender using any method you prefer.

Step 2: Wait for Transaction

Once the sender initiates the transaction:

- You don't need to do anything
- Tokens will appear in your wallet automatically
- Transaction may take a few moments to confirm

Step 3: Verify Receipt

1. Check your balance on the dashboard
2. View the transaction in your transaction history
3. Click the transaction to see details
4. Verify the amount received matches what was sent

Transaction Confirmation

Transactions typically confirm quickly, but:

- Network congestion can cause delays
- You'll see the transaction as "pending" until confirmed
- Once confirmed, balance updates automatically

Security Considerations

Address Safety

- ✅ **Safe to share**: Your wallet address (public key) is safe to share publicly
- ✅ **No risk**: Sharing your address doesn't compromise your wallet security
- ✅ **Public information**: Addresses are meant to be shared

What NOT to Share

- ❌ **Never share your Seed64**
- ❌ **Never share your password**
- ❌ **Never share your private key** (if you have access to it)

Address Verification

When receiving tokens:

- Verify the sender's identity if receiving large amounts
- Double-check the amount matches expectations
- Confirm you're receiving on the correct network (Mainnet vs Testnet)

Multiple Addresses

- Each wallet has one primary address
- You can create multiple wallets for different purposes
- Each wallet has its own unique address
- Consider using different wallets for different use cases

Network Considerations

Mainnet vs Testnet

- **Mainnet**: Real tokens with real value
- **Testnet**: Test tokens with no value (for testing)

Ensure:

- You're receiving on the correct network
- The sender is using the same network
- Your address matches the network type

Troubleshooting

Tokens not appearing

Possible causes:

- Transaction still pending

- Wrong network selected
- Transaction failed
- Address mismatch

Solutions:

- Wait a few moments for confirmation
- Check transaction status in explorer
- Verify network settings match sender's network
- Ask sender for transaction hash to verify

Wrong network

If you received tokens on the wrong network:

- Check your network settings
- Switch to the correct network
- Tokens should appear once on correct network
- Note: You cannot transfer tokens between networks directly

Address format error

If sender reports address format error:

- Verify you copied the complete address
- Ensure no extra spaces or characters
- Share QR code instead for accuracy
- Double-check address starts with correct characters

Best Practices

1. **Verify Amount**: Always verify the amount received matches expectations
2. **Check Network**: Ensure you're on the correct network
3. **Use QR Code**: QR codes prevent address entry errors

4. **Keep Records**: Save transaction details for your records
5. **Verify Sender**: For large amounts, verify sender identity

What's Next?

- View your transaction history to see received tokens
- Send tokens to others

7. Transaction History

Learn how to view and understand your transaction history in the Amadeus Wallet Extension.

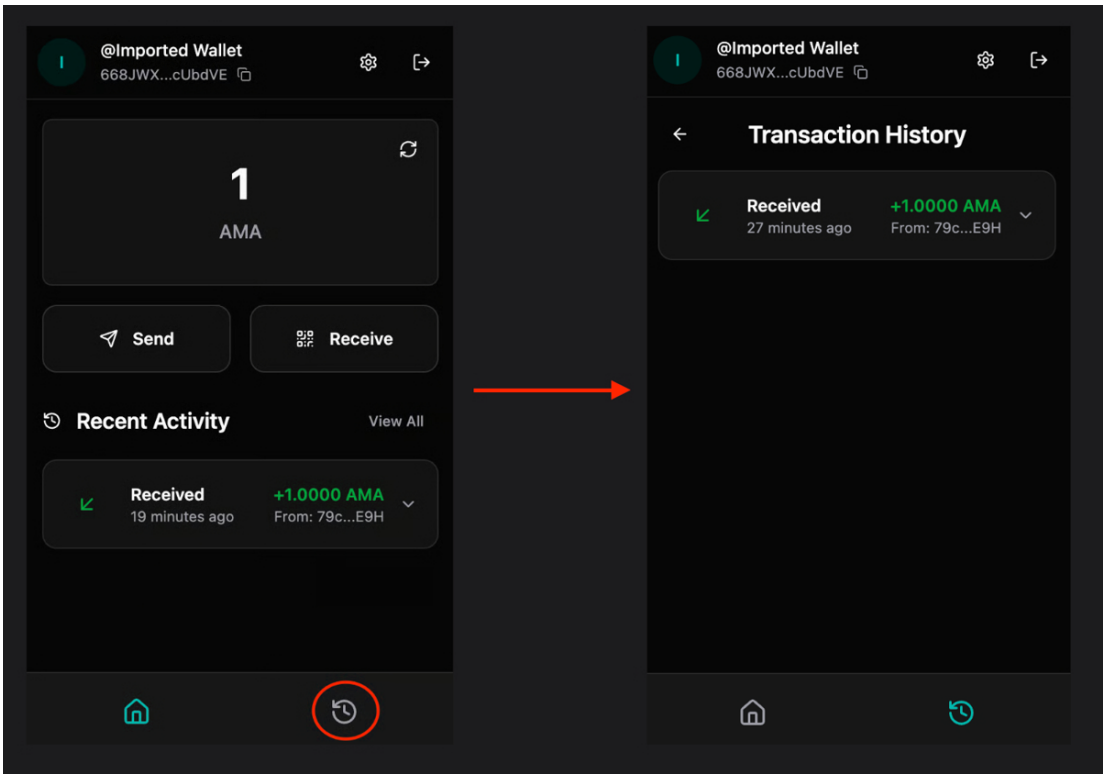
Overview

The transaction history shows all your past transactions, including sends, receives, and dApp interactions. This helps you track your token movements and verify transactions.

Accessing Transaction History

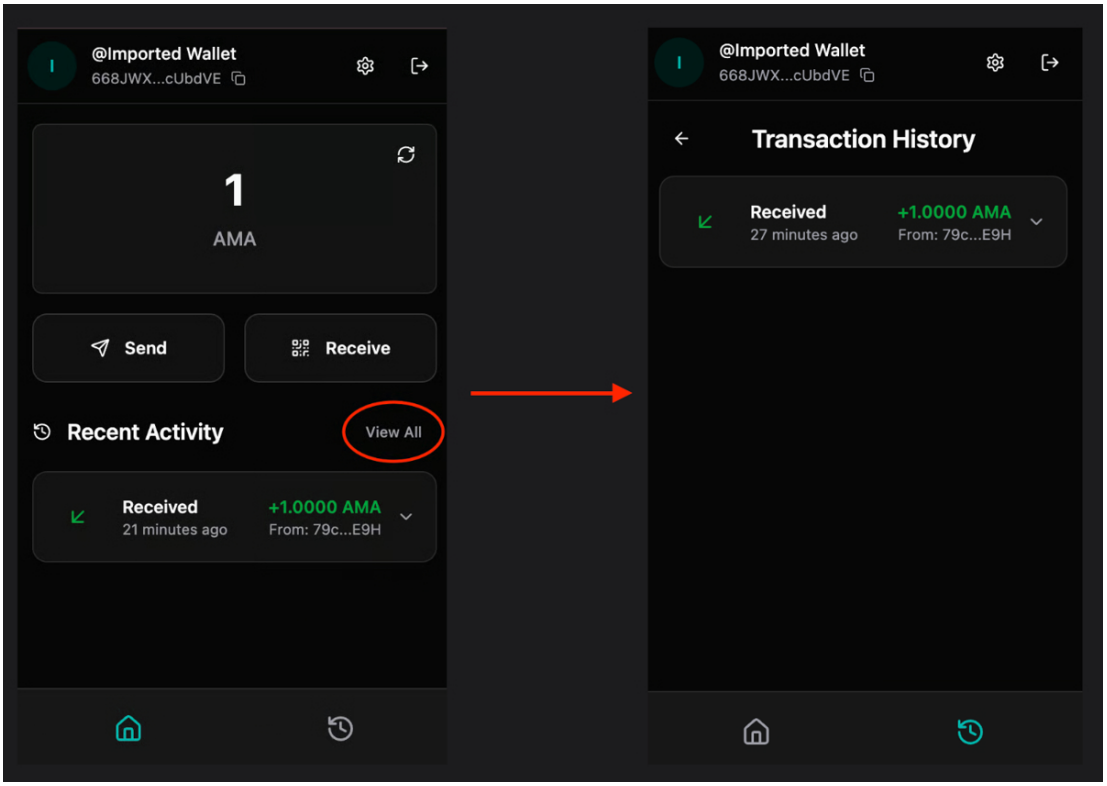
Method 1: From Navigation

1. Open the Amadeus Wallet Extension
2. Ensure your wallet is unlocked
3. Click **"History"** in the navigation menu



Method 2: From Dashboard

1. On your dashboard, scroll down to see recent transactions
2. Click **"View All"** to see complete history



Transaction List View

The history page displays transactions in a list format showing:

- **Transaction Type**: Sent or Received
- **Amount**: Token amount transferred
- **Address**: Recipient (for sends) or sender (for receives)
- **Timestamp**: When the transaction occurred
- **Status**: Success or Failed

Transaction Details

Click on any transaction to view detailed information:

Basic Information

- **Transaction Hash**: Unique identifier for the transaction

- **Status**: Success or Failed
- **Timestamp**: Exact time of the transaction
- **Block Slot**: Block number (if confirmed)

Transfer Details

- **From**: Sender's address
- **To**: Recipient's address
- **Amount**: Amount transferred
- **Token Symbol**: Token type (AMA, etc.)

Contract Information

- **Contract**: Contract address (if applicable)
- **Function**: Function called (e.g., "transfer")

Transaction Status

Success

- ✅ Green badge indicating successful transaction
- Transaction confirmed on blockchain
- Tokens successfully transferred

Failed

- ❌ Red badge indicating failed transaction
- Transaction did not complete
- May show error reason
- Tokens were not transferred

Viewing in Explorer

Each transaction can be viewed on the blockchain explorer:

1. Click the **external link icon** next to the transaction hash
2. Opens the transaction in the blockchain explorer
3. View full transaction details on the public ledger
4. Verify transaction on the blockchain

Filtering and Sorting

Sorting Options

- **Recent First** (default): Newest transactions at top
- **Oldest First**: Oldest transactions at top

Filtering

- Filter by transaction type (Sent/Received)
- Filter by status (Success/Failed)
- Search by address or transaction hash

Transaction Types

Sent Transactions

- Show outgoing token transfers
- Display recipient address
- Show negative amount (with minus sign)
- Red color indicator

Received Transactions

- Show incoming token transfers
- Display sender address
- Show positive amount (with plus sign)
- Green color indicator

dApp Transactions

- Transactions signed through dApps
- May show contract interactions
- Display contract and function details

Understanding Timestamps

Timestamps show:

- **Relative time**: "2 hours ago", "3 days ago"
- **Absolute time**: Full date and time on hover
- **Block time**: Time when transaction was included in a block

Transaction Fees

Transaction fees:

- Are deducted from your balance
- Shown in transaction details
- Vary based on network congestion
- Typically minimal amounts

Privacy Features

Hide Amounts

You can hide transaction amounts for privacy:

1. Go to **Settings** → **Preferences**
2. Toggle **"Hide Amounts"**
3. Amounts will show as "******" in transaction list

Balance Privacy

- Hide balances on dashboard
- Hide amounts in transaction history
- Useful for screenshots or screen sharing

Troubleshooting

Transactions not showing

Possible causes:

- Network mismatch (Mainnet vs Testnet)
- Wallet not synced
- Network connectivity issues

Solutions:

- Verify network settings
- Refresh the extension
- Check internet connection
- Wait for sync to complete

Wrong network transactions

If transactions appear on wrong network:

- Switch to the correct network in Settings
- Transactions are network-specific
- Mainnet and Testnet have separate histories

Missing transactions

If expected transactions don't appear:

- Verify transaction was successful
- Check transaction hash in explorer

- Ensure correct network selected
- Wait for sync to complete

Transaction details not loading

- Check internet connection
- Verify blockchain explorer is accessible
- Try refreshing the page
- Check if transaction exists in explorer

Best Practices

1. **Regular Review**: Check transaction history regularly
2. **Verify Large Transactions**: Always verify large transactions
3. **Keep Records**: Save important transaction hashes
4. **Check Status**: Verify transaction status if uncertain
5. **Use Explorer**: Use blockchain explorer for detailed verification

Security Tips

- ✅ Verify all transactions match your expectations
- ✅ Check transaction details for accuracy
- ✅ Use explorer to verify on blockchain
- ✅ Report suspicious transactions immediately
- ❌ Don't ignore failed transactions
- ❌ Don't share transaction details carelessly

What's Next?

- Send tokens to create new transactions
- Receive tokens to see incoming transactions
- Sign transactions from dApps

8. Signing Transactions

Learn how to sign transactions when interacting with decentralized applications (dApps) using the Amadeus Wallet Extension.

Overview

When you interact with dApps on the Amadeus blockchain, they may request you to sign transactions. The wallet extension provides a secure interface to review and approve these transaction requests.

How It Works

1. **dApp Request**: A dApp requests a transaction signature
2. **Extension Notification**: The extension shows a notification
3. **Review Request**: You review the transaction details
4. **Approve or Reject**: You decide to approve or reject
5. **Transaction Signed**: If approved, transaction is signed and returned

Receiving Sign Requests

Automatic Notification

When a dApp requests a transaction:

- The extension icon shows a badge notification
- Clicking the extension opens the sign request page
- You'll see the transaction details to review

Sign Request Page

The sign request page shows:

- **Origin**: Which website is requesting the signature
- **Description**: Optional description from the dApp

- **Contract**: Smart contract address
- **Method**: Function being called
- **Arguments**: Parameters for the function call

Reviewing Transaction Requests

Step 1: Review Origin

1. Check which website is requesting the signature
2. Verify it's a trusted dApp
3. Ensure the domain matches the expected dApp

Step 2: Review Details

1. **Description**: Read any description provided by the dApp
2. **Contract**: Verify the contract address is correct
3. **Method**: Understand what function is being called
4. **Arguments**: Review all parameters being passed

Step 3: Verify Safety

Before approving, verify:

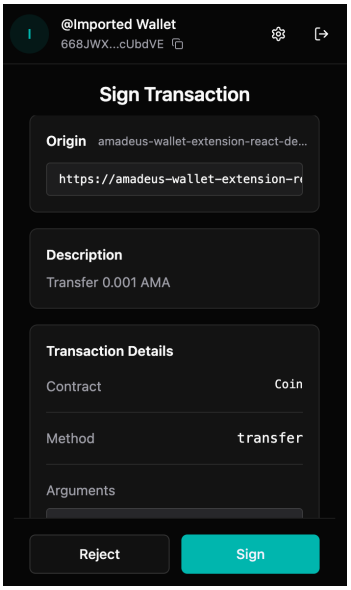
- ✅ You trust the dApp
- ✅ The transaction details are correct
- ✅ You understand what the transaction does
- ✅ The amount (if any) is correct

Approving Transactions

Step 1: Click "Sign"

1. After reviewing, click **"Sign"**
2. If your wallet is locked, you'll be prompted to unlock

3. Enter your password to unlock

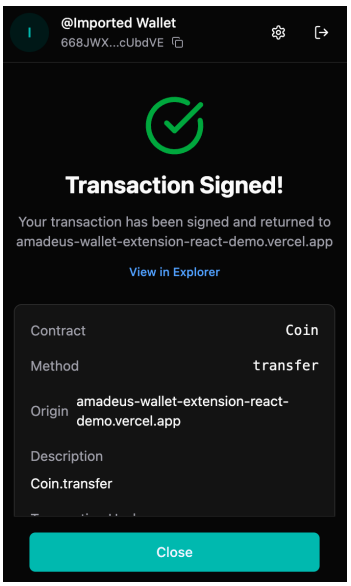


Step 2: Processing

- The extension processes your approval
- Transaction is signed with your private key
- You'll see a "Processing" screen

Step 3: Success

- Transaction is signed successfully
- Transaction hash is displayed
- You can view it in the explorer
- Transaction is returned to the dApp



Rejecting Transactions

When to Reject

Reject transactions if:

- ❌ You don't trust the dApp
- ❌ Transaction details look suspicious
- ❌ Amount or parameters are incorrect
- ❌ You didn't initiate the transaction
- ❌ Something seems wrong

How to Reject

1. Click **"Reject"** on the sign request page
2. Transaction request is cancelled
3. dApp receives rejection notification
4. No transaction is signed

Security Best Practices

Always Review

- ✅ **Always review** transaction details before approving
- ✅ **Verify origin** - ensure it's from a trusted source
- ✅ **Check amounts** - verify any token amounts
- ✅ **Understand actions** - know what the transaction does

Red Flags

Be cautious of:

- ⚠️ Requests from unknown websites
- ⚠️ Unexpected transaction requests
- ⚠️ Requests for large amounts
- ⚠️ Transactions you didn't initiate
- ⚠️ Suspicious contract addresses

Never Approve

- ❌ Transactions you don't understand
- ❌ Requests from untrusted sources
- ❌ Transactions with incorrect details
- ❌ Anything that seems suspicious

Transaction Types

Token Transfers

- **Contract**: Coin contract
- **Method**: transfer
- **Arguments**: [recipient, amount, symbol]
- **Common**: Most common transaction type

Contract Interactions

- **Contract**: Various smart contracts

- **Method**: Various functions
- **Arguments**: Contract-specific parameters
- **Use Case**: Interacting with DeFi, NFTs, etc.

Error Handling

Common Errors

"Wallet is locked"

- Unlock your wallet first
- Then approve the transaction

"Insufficient balance"

- Ensure you have enough tokens
- Check your balance before approving

"Invalid transaction"

- Transaction may be malformed
- Reject and contact dApp support

"Network mismatch"

- Ensure you're on the correct network
- Switch networks if needed

dApp Integration

For Developers

dApps integrate with the wallet using:

```
// Request accounts
const accounts = await window.amadeus.requestAccounts()

// Sign transaction
const result = await window.amadeus.signTransaction({
  contract: 'Coin',
  method: 'transfer',
  args: [recipient, amount, symbol],
  description: 'Send tokens'
})
```

See dApp Integration Guide for developers.

Troubleshooting

Sign request not appearing

Possible causes:

- Extension not installed or enabled
- dApp not properly integrated
- Browser compatibility issues

Solutions:

- Verify extension is installed and enabled
- Refresh the dApp page
- Check browser console for errors
- Ensure dApp is using correct integration

Transaction stuck

If transaction is stuck:

- Check transaction status in explorer
- Verify network connectivity
- Try rejecting and re-requesting

- Contact dApp support if persistent

Wrong transaction details

If details seem wrong:

- Reject the transaction
- Contact dApp support
- Verify you're on correct dApp
- Check for phishing attempts

Best Practices

1. **Review Carefully**: Always review transaction details
2. **Start Small**: Test with small amounts first
3. **Verify Origin**: Ensure requests are from trusted sources
4. **Understand Actions**: Know what transactions do
5. **Keep Updated**: Keep extension updated for security

What's Next?

- View signed transactions in your history
- Learn about dApp integration (for developers)
- Configure network settings for dApps

9. Network Settings

Learn how to configure network settings in the Amadeus Wallet Extension, including switching between Mainnet and Testnet, and configuring custom RPC endpoints.

Overview

The Amadeus Wallet Extension supports multiple networks:

- **Mainnet**: Production network with real tokens
- **Testnet**: Testing network with test tokens
- **Custom RPC**: Connect to custom network endpoints

Accessing Network Settings

1. Open the Amadeus Wallet Extension
2. Go to **Settings**
3. Click **"Active Networks"**

Network Options

Mainnet

Purpose: Production network for real transactions

Configuration:

- **RPC URL**: `https://nodes.amadeus.bot`
- **Explorer**: `https://explorer.ama.one`
- **Tokens**: Real AMA tokens with actual value

When to use:

- Real transactions
- Production dApps

- Actual token transfers

Testnet

Purpose: Testing network for development and testing

Configuration:

- **RPC URL**: `https://testnet-rpc.ama.one`
- **Explorer**: `https://testnet.explorer.ama.one`
- **Tokens**: Test tokens with no real value

When to use:

- Testing transactions
- Development work
- Learning and experimentation
- Trying new features

Custom RPC

Purpose: Connect to custom network endpoints

Configuration:

- **RPC URL**: Your custom endpoint
- **Explorer URL**: Optional custom explorer
- **Use Case**: Private networks, local development, custom nodes

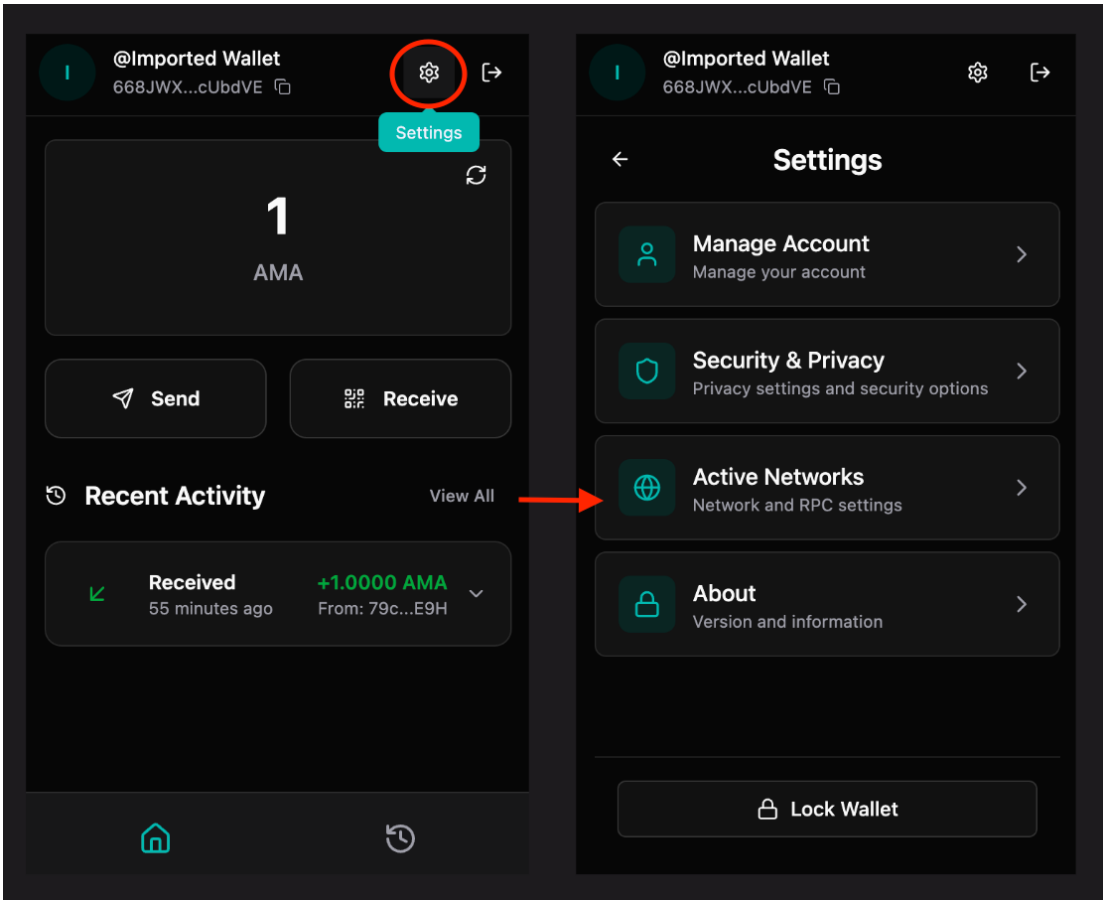
When to use:

- Local development
- Private networks
- Custom node configurations
- Testing with specific nodes

Switching Networks

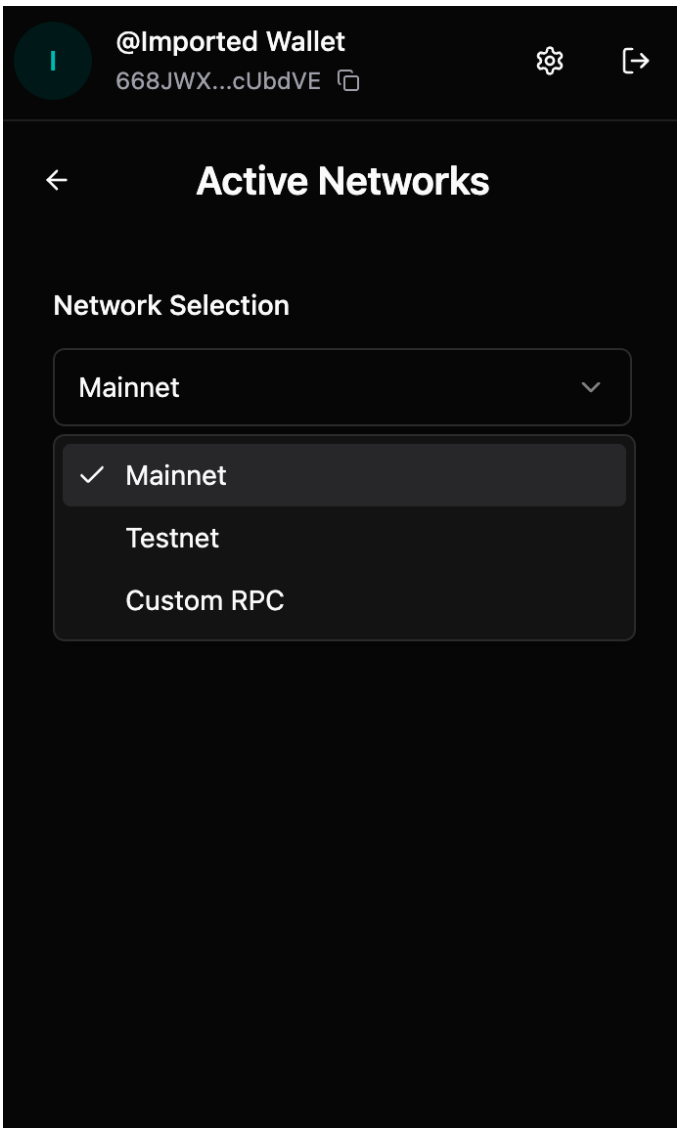
Step 1: Open Network Settings

1. Go to **Settings** → **Active Networks**
2. You'll see the network selection dropdown



Step 2: Select Network

1. Click the network dropdown
2. Select your desired network:
   - **Mainnet**
   - **Testnet**
   - **Custom RPC**



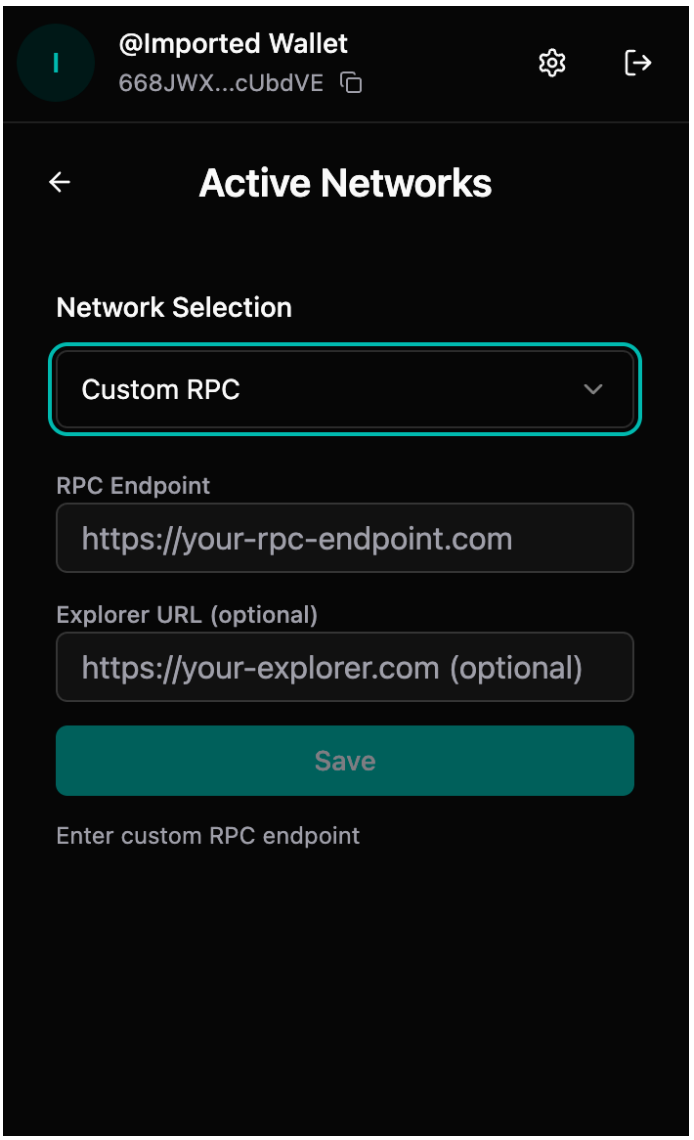
Step 3: Network Active

- Selected network becomes active immediately
- All transactions use the new network
- Balance and history sync to the new network

Configuring Custom RPC

Step 1: Select Custom RPC

1. In network settings, select **"Custom RPC"**
2. Input fields will appear for configuration



Step 2: Enter RPC Endpoint

1. **RPC Endpoint** field:
   - Enter your custom RPC URL
   - Format: `https://your-rpc-endpoint.com`
   - Must be a valid HTTP/HTTPS URL
   - Click **"Save"** when done

Step 3: Enter Explorer URL (Optional)

1. **Explorer URL** field (optional):
   - Enter custom explorer URL if available
   - Format: `https://your-explorer.com`

- Leave empty to use default explorer
- Click **"Save"** when done

Step 4: Save Configuration

1. Click **"Save"** button
2. Custom network becomes active
3. Extension connects to your custom endpoint

Network-Specific Data

Important Notes

- **Balances**: Separate for each network
- **Transaction History**: Network-specific
- **Addresses**: Same address across networks (but balances differ)
- **Tokens**: Network-specific token balances

Switching Considerations

When switching networks:

- ⚠️ Your balance will change (network-specific)
- ⚠️ Transaction history is separate per network
- ⚠️ Ensure you're on the correct network for your use case
- ⚠️ Transactions on one network don't appear on another

Verifying Network

Check Current Network

1. Go to **Settings** → **Active Networks**
2. Current network is displayed at the top
3. Network details shown below

Network Indicators

- Network name displayed in settings
- Explorer links use current network
- Transaction history filtered by network

Troubleshooting

Wrong network selected

If you're on the wrong network:

1. Go to network settings
2. Switch to correct network
3. Verify balance and transactions
4. Ensure dApp matches your network

Custom RPC not connecting

Possible causes:

- Invalid RPC URL
- Network connectivity issues
- RPC endpoint down
- CORS or security restrictions

Solutions:

- Verify RPC URL is correct and accessible
- Check internet connection
- Test RPC endpoint in browser
- Contact RPC provider if issues persist

Balance not showing

If balance doesn't appear:

- Verify network is correct
- Check internet connection
- Wait for sync to complete
- Refresh the extension
- Verify address on explorer

Transactions on wrong network

If transactions appear on wrong network:

- Switch to the correct network
- Transactions are network-specific
- Cannot transfer between networks
- Verify network before transactions

Best Practices

1. **Verify Network**: Always verify network before transactions
2. **Test First**: Use Testnet for testing
3. **Match dApps**: Ensure dApp network matches wallet network
4. **Custom RPC**: Only use trusted custom RPC endpoints
5. **Network Awareness**: Be aware which network you're on

Security Considerations

Network Security

- ✅ Use official network endpoints when possible
- ✅ Verify custom RPC endpoints are trusted
- ✅ Be cautious with custom networks
- ❌ Don't use untrusted custom RPC endpoints
- ❌ Verify network before large transactions

Phishing Prevention

- Always verify you're on the correct network
- Check network settings regularly
- Be cautious of network switching prompts
- Verify dApp network matches wallet network

Advanced Configuration

Custom Network Use Cases

- **Local Development**: Connect to local node
- **Private Networks**: Connect to private blockchain
- **Testing**: Use test networks
- **Custom Nodes**: Connect to specific node providers

RPC Endpoint Requirements

Custom RPC endpoints must:

- Support HTTP/HTTPS
- Implement Amadeus RPC API
- Be accessible from your browser
- Have proper CORS configuration

What's Next?

- Send tokens on your selected network
- View transaction history for your network

10. Integration Guide

This guide is for developers who want to integrate the Amadeus Wallet Extension into their decentralized applications (dApps).

Overview

The Amadeus Wallet Extension provides a JavaScript API that dApps can use to interact with user wallets. This allows users to sign transactions and interact with smart contracts directly from web applications.

API Reference

Checking Wallet Availability

Before using the wallet API, check if it's available:

```
if (typeof window.amadeus !== 'undefined') {
    // Wallet is available
    console.log('Amadeus wallet detected')
} else {
    // Wallet not installed
    console.log('Amadeus wallet not found')
}
```

Requesting Accounts

Request access to user's wallet accounts:

```
try {
    const accounts = await window.amadeus.requestAccounts()
    console.log('Connected accounts:', accounts)
    // Returns: ['5KJvsngHeMoo884xkJ6Cyb5StvnRN6f9tYiqwqJzLpQq']
} catch (error) {
    console.error('Failed to connect:', error.message)
}
```

Response:

- Array of account addresses (Base58 encoded)
- User must approve connection request
- Returns current account if already connected

Signing Transactions

Sign a transaction request:

```
try {
    const result = await window.amadeus.signTransaction({
        contract: 'Coin',
        method: 'transfer',
        args: [
            '5KJvsngHeMoo884xkJ6Cyb5StvnRN6f9tYiqwqJzLpQq', // recipient
            '1000000000', // amount (in atomic units)
            'AMA' // token symbol
        ],
        description: 'Send 100 AMA tokens'
    })

    console.log('Transaction signed:', result.txHash)
    console.log('Packed transaction:', result.txPacked)

    // Submit to blockchain
    // ... your submission logic
} catch (error) {
    console.error('Transaction rejected:', error.message)
}
```

Parameters:

- `contract` (string): Contract name or address
- `method` (string): Function name to call
- `args` (array): Function arguments
- `description` (string, optional): Human-readable description

Response:

- `txHash` (string): Transaction hash
- `txPacked` (Uint8Array): Packed transaction bytes

Event Listeners

Listen for wallet connection/disconnection:

```
// Listen for account changes
window.amadeus.on('accountsChanged', (accounts) => {
    console.log('Accounts changed:', accounts)
    // Update UI with new accounts
})

// Listen for disconnect
window.amadeus.on('disconnect', () => {
    console.log('Wallet disconnected')
    // Handle disconnection
})
```

Integration Examples

Basic Connection Flow

```
async function connectWallet() {
    try {
        // Check if wallet is available
        if (typeof window.amadeus === 'undefined') {
            alert('Please install Amadeus Wallet Extension')
            return
        }

        // Request accounts
        const accounts = await window.amadeus.requestAccounts()

        if (accounts.length === 0) {
            alert('No accounts found')
            return
        }

        // Use first account
        const userAddress = accounts[0]
        console.log('Connected:', userAddress)

        // Update UI
        updateUI(userAddress)
    } catch (error) {
        console.error('Connection failed:', error)
        alert('Failed to connect wallet: ' + error.message)
    }
}
```

Token Transfer Example

```
async function transferTokens(recipient, amount, symbol = 'AMA') {
    try {
        // Convert amount to atomic units
        const atomicAmount = toAtomicUnits(amount)

        // Request signature
        const result = await window.amadeus.signTransaction({
            contract: 'Coin',
            method: 'transfer',
            args: [recipient, atomicAmount.toString(), symbol],
            description: `Send ${amount} ${symbol} to ${recipient}`
        })

        // Submit transaction to blockchain
        const txResult = await submitTransaction(result.txPacked)

        console.log('Transaction submitted:', txResult.hash)
        return txResult
    } catch (error) {
        if (error.message === 'User rejected') {
            console.log('User rejected transaction')
        } else {
            console.error('Transfer failed:', error)
        }
        throw error
    }
}

// Helper function to convert to atomic units
function toAtomicUnits(amount) {
    return Math.floor(amount * 1e8) // 8 decimal places
}
```

Contract Interaction Example

```
async function callContract(contractAddress, method, args, description) {
    try {
        const result = await window.amadeus.signTransaction({
            contract: contractAddress,
            method: method,
            args: args,
            description: description || `Call ${method} on
${contractAddress}`
        })

        // Submit to blockchain
        return await submitTransaction(result.txPacked)
    } catch (error) {
        console.error('Contract call failed:', error)
        throw error
    }
}

// Example usage
await callContract('0x1234...', 'approve', ['0x5678...', '1000000000'],
'Approve token spending')
```

Error Handling

Common Errors

```
try {
  await window.amadeus.signTransaction({...})
} catch (error) {
  switch (error.message) {
    case 'User rejected':
      // User clicked reject
      console.log('Transaction rejected by user')
      break
    case 'Wallet is locked':
      // Wallet needs to be unlocked
      console.log('Please unlock your wallet')
      break
    case 'No accounts':
      // No accounts available
      console.log('Please connect your wallet')
      break
    default:
      // Other errors
      console.error('Error:', error.message)
  }
}
```

Best Practices

User Experience

1. **Check Availability First**

```
if (typeof window.amadeus === 'undefined') {
    // Show install prompt
}
```

2. **Handle Rejections Gracefully**

```
try {
  await window.amadeus.signTransaction({...})
} catch (error) {
  if (error.message === 'User rejected') {
    // Don't show error, user intentionally rejected
    return
  }
  // Show error for other cases
}
```

3. **Provide Clear Descriptions**

```
description: 'Send 100 AMA to Alice for payment'
// Better than: 'Transfer'
```

4. **Show Loading States**

```
setLoading(true)
try {
  const result = await window.amadeus.signTransaction({...})
  // Handle success
} finally {
  setLoading(false)
}
```

Security

1. **Validate Inputs**
   - Verify addresses before requesting signatures
   - Validate amounts and parameters
   - Check contract addresses
2. **Handle Errors Properly**
   - Don't expose sensitive information
   - Provide helpful error messages
   - Log errors for debugging
3. **Verify Transactions**
   - Verify transaction hash after signing
   - Check transaction status on blockchain
   - Handle failed transactions

TypeScript Support

If using TypeScript, you can define types:

```
interface AmadeusWallet {
    requestAccounts(): Promise<string[]>
    signTransaction(params: {
        contract: string
        method: string
        args: any[]
        description?: string
    }): Promise<{
        txHash: string
        txPacked: Uint8Array
    }>
    on(event: string, callback: Function): void
    removeListener(event: string, callback: Function): void
}

declare global {
    interface Window {
        amadeus?: AmadeusWallet
    }
}
```

Testing

Testnet Testing

Always test on Testnet first:

1. Ensure users are on Testnet
2. Use Testnet tokens for testing
3. Verify transactions on Testnet explorer

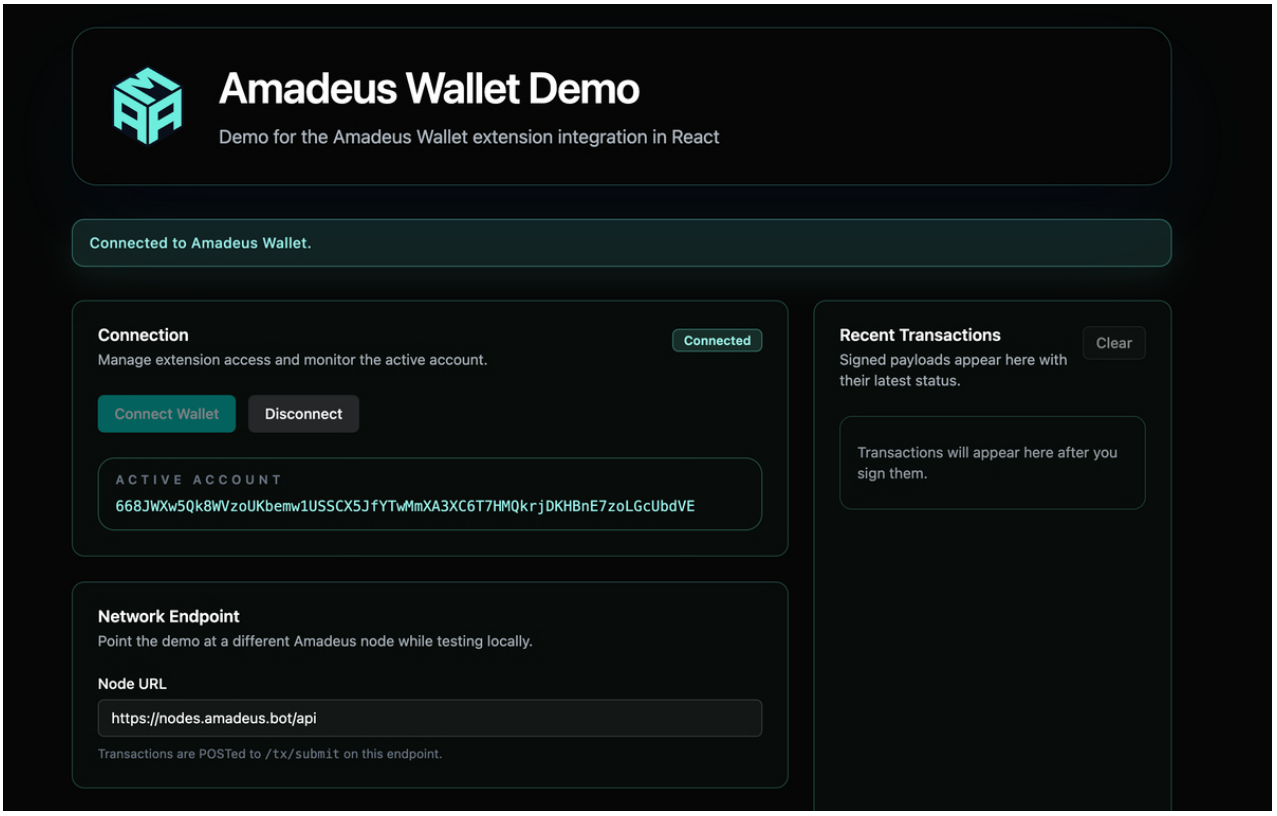
Mock Implementation

For development without extension:

```
// Mock wallet for testing
window.amadeus = {
    async requestAccounts() {
        return ['5KJvsngHeMoo884xkJ6Cyb5StvnRN6f9tYiqwqJzLpQq']
    },
    async signTransaction(params) {
        console.log('Mock signing:', params)
        return {
            txHash: 'mock-hash',
            txPacked: new Uint8Array()
        }
    },
    on() {},
    removeListener() {}
}
```

React Integration Example

We provide a complete React + TypeScript demo application that showcases how to integrate the Amadeus Wallet Extension into a modern React application.



Repository

GitHub: amadeus-wallet-extension-react-demo

Live Demo: amadeus-wallet-extension-react-demo.vercel.app

Features Demonstrated

The demo includes working examples of:

- **Provider Detection**: Automatically detects wallet availability and listens for initialization events
- **Account Management**: Connect/disconnect flows with `requestAccounts()` and `accountsChanged` event handling
- **Token Transfers**: Complete AMA token transfer flow with signing and submission
- **Custom Contract Calls**: Sign arbitrary contract calls with custom arguments
- **Transaction History**: Track signed transactions with status updates
- **Event Logging**: Real-time event logging for debugging and observability
- **Network Configuration**: Configurable API endpoint for local node testing

Tech Stack

- React 19
- TypeScript
- Vite
- Tailwind CSS
- shadcn/ui components

Getting Started

```
# Clone the repository
git clone https://github.com/amadeusprotocol/amadeus-wallet-extension-
react-demo.git
cd amadeus-wallet-extension-react-demo

# Install dependencies
bun install

# Start development server
bun run dev
```

Key Hooks and Components

The demo provides reusable React hooks:

- `useAmadeus()`: Core wallet integration hook
- `useWallet()`: Wallet connection state management
- `useTransfer()`: Token transfer operations
- `useCustomCall()`: Custom contract call handling
- `useTransactionHistory()`: Transaction tracking

These hooks can be adapted for your own React applications. See the repository's README for detailed documentation on each hook and component.

Resources

- **Official Website**: https://ama.one
- **Mainnet Explorer**: https://explorer.ama.one
- **Testnet Explorer**: https://testnet.explorer.ama.one
- **SDK Documentation**: Check @amadeus-protocol/sdk package
- **React Integration Demo**: GitHub Repository | Live Demo

Support

For integration support:

- Check extension documentation

- Review SDK documentation
- Test on Testnet first
- Report issues through official channels

Explorer

Explorer Links

Current Explorer Implementations:

- https://exporer.ama.one/
- https://ama-explorer.ddns.net/ 📙https://github.com/GerardMensoif/ama-explorer

Disclaimer: The information on this documentation is provided for informational purposes only and does not constitute financial, investment, or legal advice. Amadeus is an experimental blockchain protocol under active development; features and use cases described are illustrative only and may change or never be implemented. $AMA is a utility token designed to pay for transaction fees and network operations. It is not intended as a security or investment, and no representation is made regarding profitability, appreciation, or future value. References to “monetization,” “fees,” or other economic outcomes are hypothetical examples only. Users are solely responsible for compliance with applicable laws, including those relating to securities, data protection, and intellectual property. Amadeus and its contributors disclaim all warranties and accept no liability for losses or damages arising from the use of the protocol or deployment of AI agents. Use at your own risk.

Sdk

1. Introduction

Amadeus Protocol SDK Documentation

Welcome to the Amadeus Protocol SDK documentation. This guide will help you integrate the SDK into your applications and interact with the Amadeus blockchain.

Overview

The Amadeus Protocol SDK is a TypeScript/JavaScript library that provides a complete toolkit for building applications on the Amadeus blockchain. It offers a unified interface for interacting with nodes, building transactions, managing keys, and querying blockchain data.

Key Features

- **Complete API Client**: Full-featured HTTP client for all Amadeus node endpoints
- **Transaction Building**: Easy-to-use transaction builder with signing support
- **Cryptographic Operations**: BLS12-381 key generation, signing, and verification
- **Canonical Serialization**: VecPack encoding/decoding for deterministic data representation
- **Password-Based Encryption**: Secure AES-GCM encryption for wallet data protection
- **Type Safety**: Complete TypeScript definitions for all APIs
- **Zero Dependencies**: Uses native fetch (no external HTTP libraries)
- **Encoding Utilities**: Base58 and Base64 encoding/decoding
- **Token Conversions**: Convert between atomic units and human-readable amounts

What You Can Do

- **Query Blockchain Data**: Get chain stats, entries, transaction history
- **Manage Wallets**: Check balances, query transaction events
- **Build Transactions**: Create and sign transactions for token transfers and contract calls

- **Interact with Contracts**: Read contract data, validate bytecode, query richlists
- **Network Information**: Get peer information, validator data, epoch scores
- **Secure Storage**: Encrypt sensitive data with password-based encryption

Architecture

The SDK is organized into several key modules:

- **AmadeusSDK**: Main SDK class providing access to all modules
- **TransactionBuilder**: Class for building and signing transactions
- **API Modules**: Chain, Wallet, Transaction, Contract, Epoch, Peer, Proof
- **Utilities**: Crypto, Encoding, Serialization, Conversion, Encryption

Quick Start

```
import { AmadeusSDK, TransactionBuilder, generateKeypair } from
'@amadeus-protocol/sdk'

// Initialize SDK
const sdk = new AmadeusSDK({
    baseUrl: 'https://nodes.amadeus.bot/api'
})

// Generate a keypair
const keypair = generateKeypair()

// Build and submit a transaction
const builder = new TransactionBuilder(keypair.privateKey)
const { txHash, txPacked } = builder.transfer({
    recipient: '5Kd3N...',
    amount: 10.5,
    symbol: 'AMA'
})

const result = await sdk.transaction.submit(txPacked)
```

Documentation Structure

- **Getting Started**: Installation and basic setup
- **Core Concepts**: Understanding keypairs, addresses, transactions

- **Transaction Building**: Building and signing transactions
- **API Modules**: Complete API reference for all modules
- **Utilities**: Crypto, encoding, serialization, and conversion utilities
- **Examples**: Real-world usage examples
- **Best Practices**: Security and development best practices
- **Troubleshooting**: Common issues and solutions

Package Information

- **Package**: `@amadeus-protocol/sdk`
- **Version**: Check npm
- **License**: MIT
- **TypeScript**: Full TypeScript support with complete type definitions

Resources

- **NPM Package**: @amadeus-protocol/sdk
- **GitHub Repository**: https://github.com/amadeusprotocol/amadeus-typescript-sdk
- **Mainnet Explorer**: https://explorer.ama.one
- **Testnet Explorer**: https://testnet.explorer.ama.one

Next Steps

1. Install the SDK
2. Learn the core concepts
3. Build your first transaction
4. Explore the API modules

2. Getting Started

This guide will help you get started with the Amadeus Protocol SDK in your project. We'll cover installation, basic setup, and your first queries and transactions.

Installation

Package Installation

Install the SDK using your preferred package manager:

```
# npm
npm install @amadeus-protocol/sdk

# yarn
yarn add @amadeus-protocol/sdk

# pnpm
pnpm add @amadeus-protocol/sdk

# bun
bun add @amadeus-protocol/sdk
```

Verify Installation

After installation, verify the package is installed correctly:

```
# Check package version
npm list @amadeus-protocol/sdk

# Or check in your code
import { AmadeusSDK } from '@amadeus-protocol/sdk'
console.log('SDK Version:', AmadeusSDK.getVersion())
```

Requirements

Node.js

The SDK requires **Node.js version 20.0.0 or higher**.

Check your Node.js version:

```
node --version
```

If you need to update Node.js, visit nodejs.org.

TypeScript (Optional)

TypeScript is optional but highly recommended. The SDK includes complete type definitions.

Install TypeScript:

```
npm install -D typescript @types/node
```

Browser Support

The SDK works in modern browsers that support:

- **ES2020** features
- **fetch API** (or polyfill)
- **Web Crypto API** (for encryption features)

Basic Setup

Import the SDK

```
import { AmadeusSDK } from '@amadeus-protocol/sdk'
```

Initialize the SDK

```
// Using default node URL
const sdk = new AmadeusSDK()

// Or specify a custom node URL
const sdk = new AmadeusSDK({
    baseUrl: 'https://nodes.amadeus.bot/api',
    timeout: 30000 // Optional: request timeout in milliseconds
})
```

Your First Query

Let's start with a simple chain query:

```
import { AmadeusSDK } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()

// Get the current chain tip
const tip = await sdk.chain.getTip()
console.log('Current height:', tip.entry.height)
console.log('Chain hash:', tip.entry.hash)
```

Your First Transaction

Here's how to build and submit a simple token transfer:

```
import { AmadeusSDK, TransactionBuilder, generateKeypair } from
'@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()

// Generate a keypair (or use an existing one)
const keypair = generateKeypair()
console.log('Public Key (Address):', keypair.publicKey)
console.log('Private Key (Seed):', keypair.privateKey) // Keep this
secret!

// Create a transaction builder
const builder = new TransactionBuilder(keypair.privateKey)

// Build and sign a transfer transaction
const { txHash, txPacked } = builder.transfer({
    recipient: '5Kd3N...', // Recipient's address
    amount: 10.5, // Amount in AMA
    symbol: 'AMA' // Token symbol
})

console.log('Transaction hash:', txHash)

// Submit the transaction
const result = await sdk.transaction.submit(txPacked)
if (result.error === 'ok') {
    console.log('Transaction submitted successfully!')
} else {
    console.error('Transaction error:', result.error)
}
```

TypeScript Configuration

If using TypeScript, ensure your `tsconfig.json` includes:

```
{
    "compilerOptions": {
        "target": "ES2020",
        "module": "ESNext",
        "lib": ["ES2020"],
        "moduleResolution": "node",
        "esModuleInterop": true,
        "strict": true
    }
}
```

Common Imports

Here are the most commonly used imports:

```
// Main SDK class
import { AmadeusSDK } from '@amadeus-protocol/sdk'

// Transaction building
import { TransactionBuilder } from '@amadeus-protocol/sdk'

// Key generation
import { generateKeypair, derivePublicKeyFromSeedBase58 } from '@amadeus-
protocol/sdk'

// Encoding utilities
import { toBase58, fromBase58 } from '@amadeus-protocol/sdk'

// Amount conversions
import { toAtomicAma, fromAtomicAma } from '@amadeus-protocol/sdk'

// Encryption
import { encryptWithPassword, decryptWithPassword } from '@amadeus-
protocol/sdk'
```

Troubleshooting Installation

"Module not found" Error

Problem: Cannot find module '@amadeus-protocol/sdk'

Solutions:

1. Ensure the package is installed:

   ```
   npm install @amadeus-protocol/sdk
   ```

2. Check your `package.json`:

```
{
    "dependencies": {
        "@amadeus-protocol/sdk": "^1.0.0"
    }
}
```

3. Clear cache and reinstall:

```
npm cache clean --force
rm -rf node_modules package-lock.json
npm install
```

TypeScript Errors

Problem: Type errors when importing

Solutions:

1. Ensure TypeScript is installed:

```
npm install -D typescript
```

2. Check `tsconfig.json`:

```
{
    "compilerOptions": {
        "moduleResolution": "node",
        "esModuleInterop": true
    }
}
```

3. Restart your TypeScript server/IDE

Node.js Version Issues

Problem: Errors related to Node.js version

Solutions:

1. Check Node.js version:

```
node --version
```

2. Update to Node.js 20+ if needed
3. Use a version manager like `nvm`:

```
nvm install 20
nvm use 20
```

Next Steps

Now that you have the SDK installed and running:

1. **Learn Core Concepts**: Understand keypairs, addresses, and transactions
2. **Build Transactions**: Learn how to create and sign transactions
3. **Explore API Modules**: Discover all available API endpoints
4. **Check Examples**: See real-world usage examples

3. Core Concepts

Understanding the fundamental concepts of the Amadeus Protocol SDK.

Keypairs

Amadeus Protocol uses **BLS12-381** cryptography for all cryptographic operations. Each account has a public/private keypair.

Generating Keypairs

```
import { generateKeypair } from '@amadeus-protocol/sdk'

// Generate a new keypair
const keypair = generateKeypair()

console.log('Public Key:', keypair.publicKey) // Base58 encoded
console.log('Private Key:', keypair.privateKey) // Base58 encoded seed
(64 bytes)
```

Keypair Structure

- **Public Key**: 48-byte BLS12-381 public key, Base58-encoded
- **Private Key**: 64-byte seed, Base58-encoded (also called "Seed64")

Deriving Public Keys

```
import { derivePublicKeyFromSeedBase58 } from '@amadeus-protocol/sdk'

// Derive public key from existing seed
const publicKey = derivePublicKeyFromSeedBase58('5Kd3N...')
console.log('Public Key:', publicKey)
```

Addresses

In Amadeus Protocol, **public keys serve as addresses**. They are Base58-encoded 48-byte BLS12-381 public keys.

Address Format

```
// Address is the public key
const address = keypair.publicKey
// Example: "5Kd3NvsngHeMoo884xkJ6Cyb5StvnRN6f9tYiqwqJzLpQq"
```

Address Characteristics

- **Length**: Typically 87-88 characters (Base58 encoded)
- **Format**: Base58-encoded public key
- **Uniqueness**: Each keypair has a unique address
- **Public**: Safe to share publicly (it's your public key)

Address Validation

```
import { fromBase58 } from '@amadeus-protocol/sdk'

function isValidAddress(address: string): boolean {
    try {
        const bytes = fromBase58(address)
        return bytes.length === 48 // BLS12-381 public key is 48 bytes
    } catch {
        return false
    }
}
```

Transactions

Transactions are the primary way to interact with the Amadeus blockchain. They must be:

1. **Built** with proper structure
2. **Signed** with the sender's private key
3. **Serialized** (packed) into binary format
4. **Submitted** to a node

Transaction Structure

```
interface UnsignedTransaction {
    signer: Uint8Array // Sender's public key (48 bytes)
    nonce: bigint // Transaction nonce
    action: TransactionAction // Transaction action
}

interface TransactionAction {
    op: 'call' // Operation type
    contract: string // Contract name or address
    function: string // Function to call
    args: SerializableValue[] // Function arguments
}
```

Transaction Lifecycle

```
import { TransactionBuilder } from '@amadeus-protocol/sdk'

const builder = new TransactionBuilder(privateKey)

// 1. Build unsigned transaction
const unsignedTx = builder.buildTransfer({
    recipient: address,
    amount: 10.5,
    symbol: 'AMA'
})

// 2. Sign the transaction
const { txHash, txPacked } = builder.sign(unsignedTx)

// 3. Submit to blockchain
const result = await sdk.transaction.submit(txPacked)
```

Transaction Nonces

Nonces are automatically generated using timestamps:

```
// Nonce is generated as: BigInt(Date.now()) * 1_000_000n
// This ensures uniqueness for transactions
```

For high-frequency transactions, ensure sufficient time between transactions to avoid nonce collisions.

Amounts

AMA tokens use **9 decimal places**. Always use atomic units for transactions.

Converting Amounts

```
import { toAtomicAma, fromAtomicAma } from '@amadeus-protocol/sdk'

// Convert to atomic units (for transactions)
const atomic = toAtomicAma(1.5) // Returns 1500000000

// Convert from atomic units (for display)
const human = fromAtomicAma(1500000000) // Returns 1.5
```

Amount Precision

Always use `toAtomicAma` when building transactions:

```
// ✅ Good
const amount = toAtomicAma(1.5)

// ❌ Bad - may lose precision
const amount = 1.5 * 1000000000
```

Serialization

The SDK uses **VecPack** canonical serialization for deterministic encoding of transaction data.

Encoding

```
import { encode } from '@amadeus-protocol/sdk'

const data = {
    foo: 'bar',
    count: 42,
    items: [1, 2, 3]
}

const encoded = encode(data) // Returns Uint8Array
```

Decoding

```
import { decode } from '@amadeus-protocol/sdk'

const decoded = decode(encoded) // Returns DecodedValue
```

Supported Types

- `null`
- `boolean`
- `number` / `bigint`
- `string`
- `Uint8Array`
- Arrays
- Objects / Maps

Signing

Transactions are signed using BLS12-381 signatures over the transaction hash.

Signing Process

1. Build transaction structure
2. Serialize transaction (canonical encoding)
3. Hash the serialized transaction (SHA-256)
4. Sign the hash with private key (BLS12-381)

5. Pack transaction with signature

```
// The SDK handles all of this automatically
const { txHash, txPacked } = builder.sign(unsignedTx)
```

Encoding Formats

Base58

Used for addresses, keys, and transaction hashes:

```
import { toBase58, fromBase58 } from '@amadeus-protocol/sdk'

// Encode bytes to Base58
const encoded = toBase58(new Uint8Array([1, 2, 3]))

// Decode Base58 to bytes
const decoded = fromBase58('5Kd3N...')
```

Base64

Used for encrypted data and binary payloads:

```
import { uint8ArrayToBase64, base64ToUint8Array } from '@amadeus-
protocol/sdk'

// Encode to Base64
const base64 = uint8ArrayToBase64(bytes)

// Decode from Base64
const bytes = base64ToUint8Array(base64)
```

Encryption

The SDK provides password-based encryption for securing sensitive data.

Encrypting Data

```
import { encryptWithPassword } from '@amadeus-protocol/sdk'

// Encrypt private key before storage
const encrypted = await encryptWithPassword(privateKey, userPassword)
// Returns: { encryptedData, iv, salt } (all Base64 encoded)
```

Decrypting Data

```
import { decryptWithPassword } from '@amadeus-protocol/sdk'

// Decrypt when needed
const decrypted = await decryptWithPassword(encrypted, userPassword)
```

Security Features

- **AES-GCM** encryption with 256-bit keys
- **PBKDF2** key derivation with 100,000 iterations
- Unique salt and IV for each encryption
- Authenticated encryption (prevents tampering)

API Client

The SDK provides a unified API client for interacting with Amadeus nodes.

Initialization

```
import { AmadeusSDK } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK({
    baseUrl: 'https://nodes.amadeus.bot/api',
    timeout: 30000
})
```

API Modules

- `sdk.chain` - Chain queries (tip, stats, entries)
- `sdk.wallet` - Wallet operations (balances, transactions)

- `sdk.transaction` - Transaction submission
- `sdk.contract` - Contract interactions
- `sdk.epoch` - Epoch and validator data
- `sdk.peer` - Network peer information
- `sdk.proof` - Validator proofs

Error Handling

All SDK errors are instances of `AmadeusSDKError`:

```
import { AmadeusSDKError } from '@amadeus-protocol/sdk'

try {
    const balance = await sdk.wallet.getBalance(address, 'AMA')
} catch (error) {
    if (error instanceof AmadeusSDKError) {
        console.error('SDK Error:', error.message)
        console.error('Status:', error.status)
        console.error('Response:', error.response)
    }
}
```

Next Steps

Now that you understand the core concepts:

1. **Transaction Building**: Learn how to build and sign transactions
2. **API Modules**: Explore all available API endpoints
3. **Utilities**: Discover utility functions
4. **Examples**: See real-world usage examples

4. Transaction Building

Complete guide to building and signing transactions with the Amadeus Protocol SDK.

Overview

Transactions in Amadeus Protocol are structured data that represent actions on the blockchain. They must be properly built, signed, and serialized before submission.

TransactionBuilder Class

The `TransactionBuilder` class provides a convenient API for building and signing transactions.

Creating a Builder Instance

```
import { TransactionBuilder } from '@amadeus-protocol/sdk'

// Create builder with private key (convenient for multiple transactions)
const builder = new TransactionBuilder('5Kd3N...') // Base58 encoded seed

// Or create without private key (must provide keys in each method call)
const builder = new TransactionBuilder()
```

Building Transfer Transactions

Option 1: Build and Sign in One Step (Recommended)

```
const builder = new TransactionBuilder(privateKey)

const { txHash, txPacked } = builder.transfer({
    recipient: '5Kd3N...', // Base58 encoded recipient address
    amount: 10.5, // Amount in human-readable format
    symbol: 'AMA' // Token symbol
})

// Submit the transaction
const result = await sdk.transaction.submit(txPacked)
```

Option 2: Build Unsigned, Then Sign (More Control)

```
const builder = new TransactionBuilder(privateKey)

// Build unsigned transaction
const unsignedTx = builder.buildTransfer({
    recipient: '5Kd3N...',
    amount: 10.5,
    symbol: 'AMA'
})

// Can inspect or modify unsignedTx before signing
console.log('Nonce:', unsignedTx.tx.nonce)
console.log('Action:', unsignedTx.tx.action)

// Sign the transaction
const { txHash, txPacked } = builder.sign(unsignedTx)

// Submit the transaction
const result = await sdk.transaction.submit(txPacked)
```

Building Custom Transactions

Option 1: Build and Sign in One Step

```
import { TransactionBuilder, fromBase58, toAtomicAma } from '@amadeus-
protocol/sdk'

const builder = new TransactionBuilder(privateKey)

const { txHash, txPacked } = builder.buildAndSign('Coin', 'transfer', [
    fromBase58('5Kd3N...'), // Recipient bytes
    toAtomicAma(10.5).toString(), // Amount in atomic units
    'AMA' // Token symbol
])
```

Option 2: Build Unsigned, Then Sign

```
const builder = new TransactionBuilder(privateKey)

// Build unsigned transaction
const unsignedTx = builder.build('Coin', 'transfer', [
    fromBase58('5Kd3N...'),
    toAtomicAma(10.5).toString(),
    'AMA'
])

// Inspect or modify if needed
console.log('Transaction:', unsignedTx.tx)

// Sign the transaction
const { txHash, txPacked } = builder.sign(unsignedTx)
```

Using Static Methods

You can also use static methods without creating an instance:

Build and Sign Transfer

```
import { TransactionBuilder } from '@amadeus-protocol/sdk'

const { txHash, txPacked } = TransactionBuilder.buildSignedTransfer({
    senderPrivkey: '5Kd3N...', // Base58 encoded seed
    recipient: '5Kd3N...', // Base58 encoded recipient address
    amount: 10.5, // Amount in human-readable format
    symbol: 'AMA' // Token symbol
})
```

Build Unsigned, Then Sign

```
import {
    TransactionBuilder,
    getPublicKey,
    deriveSkAndSeed64FromBase58Seed
} from '@amadeus-protocol/sdk'

// Derive keys from seed
const { seed64 } = deriveSkAndSeed64FromBase58Seed('5Kd3N...')
const signerPubKey = getPublicKey(seed64)

// Build unsigned transfer
const unsignedTx = TransactionBuilder.buildTransfer(
    { recipient: '5Kd3N...', amount: 10.5, symbol: 'AMA' },
    signerPubKey
)

// Sign the transaction
const { txHash, txPacked } = TransactionBuilder.sign(unsignedTx,
'5Kd3N...')
```

Transaction Structure

Unsigned Transaction

```
interface UnsignedTransaction {
    signer: Uint8Array // Sender's public key (48 bytes)
    nonce: bigint // Transaction nonce
    action: TransactionAction
}

interface TransactionAction {
    op: 'call' // Operation type
    contract: string // Contract name or address
    function: string // Function to call
    args: SerializableValue[] // Function arguments
}
```

Signed Transaction Result

```
interface BuildTransactionResult {
    txHash: string // Transaction hash (Base58 encoded)
    txPacked: Uint8Array // Packed transaction ready for submission
}
```

Transaction Nonces

Nonces are automatically generated using timestamps:

```
// Nonce = BigInt(Date.now()) * 1_000_000n
// This ensures uniqueness for transactions
```

Important: For high-frequency transactions, ensure sufficient time between transactions to avoid nonce collisions.

Amount Handling

Always use atomic units for transaction amounts:

```
import { toAtomicAma } from '@amadeus-protocol/sdk'

// ✅ Good - use conversion function
const amount = toAtomicAma(10.5) // Returns 10500000000

// ❌ Bad - may lose precision
const amount = 10.5 * 1000000000
```

Submitting Transactions

Submit Without Waiting

```
const result = await sdk.transaction.submit(txPacked)

if (result.error === 'ok') {
    console.log('Transaction hash:', result.hash)
} else {
    console.error('Transaction error:', result.error)
}
```

Submit and Wait for Confirmation

```
try {
    const result = await sdk.transaction.submitAndWait(txPacked)

    if (result.error === 'ok') {
        console.log('Transaction confirmed!')
        console.log('Hash:', result.hash)
        console.log('Entry hash:', result.metadata?.entry_hash)
        console.log('Receipt:', result.receipt)
    } else {
        console.error('Transaction error:', result.error)
    }
} catch (error) {
    console.error('Transaction failed or timed out:', error)
}
```

Common Transaction Types

Token Transfer

```
const builder = new TransactionBuilder(privateKey)

const { txHash, txPacked } = builder.transfer({
    recipient: recipientAddress,
    amount: amount,
    symbol: 'AMA'
})
```

Contract Call

```
const builder = new TransactionBuilder(privateKey)

const { txHash, txPacked } = builder.buildAndSign('ContractName',
'functionName', [
    arg1,
    arg2,
    arg3
])
```

Error Handling

```
import { AmadeusSDKError } from '@amadeus-protocol/sdk'

try {
    const { txHash, txPacked } = builder.transfer({
        recipient: address,
        amount: 10.5,
        symbol: 'AMA'
    })

    const result = await sdk.transaction.submit(txPacked)

    if (result.error === 'ok') {
        console.log('Success:', result.hash)
    } else {
        console.error('Transaction error:', result.error)
    }
} catch (error) {
    if (error instanceof AmadeusSDKError) {
        console.error('SDK Error:', error.message)
    } else {
        console.error('Unexpected error:', error)
    }
}
```

Best Practices

1. Always Validate Inputs

```
function validateAddress(address: string): boolean {
    try {
        const bytes = fromBase58(address)
        return bytes.length === 48
    } catch {
        return false
    }
}

if (!validateAddress(recipient)) {
    throw new Error('Invalid recipient address')
}
```

2. Check Balance Before Transferring

```
const balance = await sdk.wallet.getBalance(senderAddress, 'AMA')
if (balance.balance.float < amount + fee) {
    throw new Error('Insufficient balance')
}
```

3. Use Appropriate Nonce Timing

For high-frequency transactions, add delays:

```
async function submitMultipleTransactions(txs: Transaction[]) {
    for (const tx of txs) {
        await sdk.transaction.submit(tx)
        await new Promise((resolve) => setTimeout(resolve, 100)) // 100ms
delay
    }
}
```

4. Handle Transaction Errors

```
const result = await sdk.transaction.submit(txPacked)

switch (result.error) {
    case 'ok':
        console.log('Success')
        break
    case 'insufficient_funds':
        console.error('Not enough balance')
        break
    case 'invalid_signature':
        console.error('Invalid signature')
        break
    default:
        console.error('Unknown error:', result.error)
}
```

Advanced Usage

Building Transactions Without Private Key

```
const builder = new TransactionBuilder()

// Must provide public key for building
const { seed64 } = deriveSkAndSeed64FromBase58Seed(privateKey)
const signerPubKey = getPublicKey(seed64)

// Build unsigned transaction
const unsignedTx = builder.build('Coin', 'transfer', args, signerPubKey)

// Must provide private key for signing
const { txHash, txPacked } = builder.sign(unsignedTx, privateKey)
```

Inspecting Transactions

```
const unsignedTx = builder.buildTransfer({
    recipient: address,
    amount: 10.5,
    symbol: 'AMA'
})

// Inspect transaction details
console.log('Signer:', toBase58(unsignedTx.tx.signer))
console.log('Nonce:', unsignedTx.tx.nonce.toString())
console.log('Contract:', unsignedTx.tx.action.contract)
console.log('Function:', unsignedTx.tx.action.function)
console.log('Args:', unsignedTx.tx.action.args)
console.log('Hash:', toBase58(unsignedTx.hash))
```

Next Steps

- **API Modules**: Learn about all available API endpoints
- **Examples**: See complete transaction examples
- **Best Practices**: Security and development guidelines

5. API Modules

Complete reference for all API modules in the Amadeus Protocol SDK.

Overview

The SDK provides access to various API modules through the `AmadeusSDK` instance:

```
import { AmadeusSDK } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()

// Access API modules
sdk.chain // Chain API
sdk.wallet // Wallet API
sdk.transaction // Transaction API
sdk.contract // Contract API
sdk.epoch // Epoch API
sdk.peer // Peer API
sdk.proof // Proof API
```

Chain API

Query blockchain data including chain tip, statistics, entries, and transaction events.

`getTip()`

Get the current chain tip (latest entry).

```
const { entry } = await sdk.chain.getTip()

console.log('Height:', entry.height)
console.log('Hash:', entry.hash)
console.log('Timestamp:', entry.timestamp)
```

Returns: `Promise<GetTipResponse>` with chain entry

`getStats()`

Get chain statistics.

```
const { stats } = await sdk.chain.getStats()

console.log('Total entries:', stats.total_entries)
console.log('Total transactions:', stats.total_transactions)
console.log('Current height:', stats.height)
console.log('Circulating supply:', stats.circulating)
```

Returns: `Promise<GetStatsResponse>` with chain statistics

`getByHeight(height: number)`

Get entries at a specific height.

```
const { entries } = await sdk.chain.getByHeight(1000)

entries.forEach((entry) => {
    console.log('Entry hash:', entry.hash)
    console.log('Height:', entry.height)
})
```

Parameters:

- `height` (number): Chain height

Returns: `Promise<GetByHeightResponse>` with array of entries

`getByHash(hash: string)`

Get entries by hash.

```
const { entry } = await sdk.chain.getByHash('5Kd3N...')

console.log('Entry:', entry)
```

Parameters:

- `hash` (string): Entry hash (Base58 encoded)

Returns: `Promise<GetByHashResponse>` with chain entry

`getTransactionEventsByAccount(address: string, filters?: TransactionFilters)`

Get transaction events for an account.

```
const events = await sdk.chain.getTransactionEventsByAccount('5Kd3N...',
{
    limit: 10,
    sort: 'desc',
    type: 'sent' // or 'recv'
})

events.txs.forEach((event) => {
    console.log('Type:', event.type)
    console.log('Amount:', event.amount)
    console.log('Symbol:', event.symbol)
    console.log('Timestamp:', event.timestamp)
})
```

Parameters:

- `address` (string): Account address (Base58 encoded)
- `filters` (optional): Transaction filters

Returns: `Promise<GetTransactionEventsByAccountResponse>` with transaction events

Wallet API

Query wallet balances and transaction information.

`getBalance(address: string, symbol?: string)`

Get balance for a specific address and token symbol.

```
const balance = await sdk.wallet.getBalance('5Kd3N...', 'AMA')

console.log('Balance (float):', balance.balance.float)
console.log('Balance (flat):', balance.balance.flat)
console.log('Symbol:', balance.balance.symbol)
```

Parameters:

- `address` (string): Wallet address (Base58 encoded)
- `symbol` (string, optional): Token symbol (default: 'AMA')

Returns: `Promise<WalletBalance>` with token balance

`getAllBalances(address: string)`

Get all token balances for an address.

```
const { balances } = await sdk.wallet.getAllBalances('5Kd3N...')

Object.entries(balances).forEach(([symbol, balance]) => {
    console.log(`${symbol}: ${balance.float}`)
})
```

Parameters:

- `address` (string): Wallet address (Base58 encoded)

Returns: `Promise<GetAllBalancesResponse>` with all balances

`getTransactions(address: string, query?: GetWalletTransactionsQuery)`

Get transactions for a wallet address.

```
const { txs, cursor } = await sdk.wallet.getTransactions('5Kd3N...', {
    limit: 10,
    sort: 'desc',
    contract: 'Coin',
    function: 'transfer'
})

txs.forEach((tx) => {
    console.log('Hash:', tx.hash)
    console.log('Action:', tx.tx.action)
})
```

Parameters:

- `address` (string): Wallet address (Base58 encoded)
- `query` (optional): Query parameters (limit, sort, contract, function, type, cursor)

Returns: `Promise<GetWalletTransactionsResponse>` with transactions and cursor

Transaction API

Submit transactions and query transaction data.

`submit(txPacked: Uint8Array | string)`

Submit a transaction to the network.

```
const result = await sdk.transaction.submit(txPacked)

if (result.error === 'ok') {
    console.log('Transaction hash:', result.hash)
} else {
    console.error('Error:', result.error)
}
```

Parameters:

- `txPacked` (Uint8Array | string): Packed transaction as Uint8Array or Base58 string

Returns: `Promise<SubmitTransactionResponse>` with submission result

`submitAndWait(txPacked: Uint8Array | string)`

Submit a transaction and wait for confirmation.

```
try {
    const result = await sdk.transaction.submitAndWait(txPacked)

    if (result.error === 'ok') {
        console.log('Hash:', result.hash)
        console.log('Entry hash:', result.metadata?.entry_hash)
        console.log('Receipt:', result.receipt)
    }
} catch (error) {
    console.error('Transaction failed or timed out:', error)
}
```

Parameters:

- `txPacked` (Uint8Array | string): Packed transaction

Returns: `Promise<SubmitAndWaitTransactionResponse>` with confirmation result

`get(txHash: string)`

Get a transaction by hash.

```
const tx = await sdk.transaction.get('5Kd3N...')

console.log('Transaction:', tx.hash)
console.log('Signer:', tx.tx.signer)
console.log('Action:', tx.tx.action)
console.log('Receipt:', tx.receipt)
```

Parameters:

- `txHash` (string): Transaction hash (Base58 encoded)

Returns: `Promise<Transaction>` with transaction data

`getTransactionsInEntry(entryHash: string)`

Get all transactions in a specific entry.

```
const { txs } = await sdk.transaction.getTransactionsInEntry('5Kd3N...')

txs.forEach((tx) => {
    console.log('Transaction:', tx.hash)
})
```

Parameters:

- `entryHash` (string): Entry hash (Base58 encoded)

Returns: `Promise<GetTransactionsInEntryResponse>` with transactions

Contract API

Interact with smart contracts and query contract data.

`get(key: Uint8Array | string)`

Get contract data by key.

```
const data = await sdk.contract.get(keyBytes)
// or
const data = await sdk.contract.get('5Kd3N...') // Base58 encoded key

console.log('Contract data:', data)
```

Parameters:

- `key` (Uint8Array | string): Contract key as Uint8Array or Base58 string

Returns: `Promise<ContractDataValue>` with contract data

`getPrefix(key: Uint8Array | string)`

Get contract data by key prefix.

```
const data = await sdk.contract.getPrefix(keyPrefix)

data.forEach((item) => {
    console.log('Key:', item.key)
    console.log('Value:', item.value)
})
```

Parameters:

- `key` (Uint8Array | string): Contract key prefix

Returns: `Promise<ContractDataValue[]>` with array of key-value pairs

`validateBytecode(bytecode: Uint8Array | ArrayBuffer)`

Validate contract bytecode.

```
const result = await sdk.contract.validateBytecode(wasmBytecode)

if (result.error === 'ok') {
    console.log('Bytecode is valid')
} else {
    console.error('Validation error:', result.error)
}
```

Parameters:

- `bytecode` (Uint8Array | ArrayBuffer): Contract bytecode to validate

Returns: `Promise<ValidateBytecodeResponse>` with validation result

`getRichlist()`

Get contract richlist (token holders).

```
const { richlist } = await sdk.contract.getRichlist()

richlist.forEach((entry) => {
    console.log('Address:', entry.address)
    console.log('Balance:', entry.balance)
    console.log('Rank:', entry.rank)
})
```

Returns: `Promise<GetRichlistResponse>` with richlist entries

Epoch API

Query epoch and validator data.

`getScore(publicKey?: string)`

Get epoch score(s).

```
// Get score for specific validator
const score = await sdk.epoch.getScore('5Kd3N...')
console.log('Score:', score.score)

// Get all scores
const allScores = await sdk.epoch.getScore()
allScores.forEach(([address, score]) => {
    console.log(`${address}: ${score}`)
})
```

Parameters:

- `publicKey` (string, optional): Validator public key (Base58 encoded)

Returns: `Promise<EpochScore | EpochScore[]>` with score(s)

`getAllScores()`

Get all epoch scores.

```
const scores = await sdk.epoch.getAllScores()

scores.forEach(([address, score]) => {
    console.log(`${address}: ${score}`)
})
```

Returns: `Promise<EpochScore[]>` with all scores

`getTopValidators(limit: number)`

Get top validators by score.

```
const topValidators = await sdk.epoch.getTopValidators(10)

topValidators.forEach(([address, score]) => {
    console.log(`${address}: ${score}`)
})
```

Parameters:

- `limit` (number): Number of top validators to return

Returns: `Promise<EpochScore[]>` with top validators

`getEmissionAddress()`

Get the current emission address.

```
const { emission_address } = await sdk.epoch.getEmissionAddress()

if (emission_address) {
    console.log('Emission address:', emission_address)
} else {
    console.log('No emission address set')
}
```

Returns: `Promise<GetEmissionAddressResponse>` with emission address

`getSolInEpoch(epoch: number, solutionHash: string)`

Check if a solution is in a specific epoch.

```
const result = await sdk.epoch.getSolInEpoch(100, '5Kd3N...')

if (result.error === 'ok') {
    console.log('Solution found in epoch')
} else if (result.error === 'sol_not_found') {
    console.log('Solution not found')
} else {
    console.log('Invalid epoch')
}
```

Parameters:

- `epoch` (number): Epoch number
- `solutionHash` (string): Solution hash (Base58 encoded)

Returns: `Promise<GetSolInEpochResponse>` with check result

Peer API

Query network peer information.

`getNodes()`

Get all network nodes.

```
const nodes = await sdk.peer.getNodes()

nodes.forEach((node) => {
    console.log('Public Key:', node.pk)
    console.log('Version:', node.version)
    console.log('Height:', node.temporal_height)
    console.log('Online:', node.online)
})
```

Returns: `Promise<PeerInfo[]>` with node information

`getTrainers()`

Get all trainers.

```
const trainers = await sdk.peer.getTrainers()

trainers.forEach((trainer) => {
    console.log('Trainer:', trainer.pk)
    console.log('Is Trainer:', trainer.is_trainer)
})
```

Returns: `Promise<PeerInfo[]>` with trainer information

`getRemovedTrainers()`

Get removed trainers.

```
const removed = await sdk.peer.getRemovedTrainers()

removed.forEach((trainer) => {
    console.log('Removed trainer:', trainer.pk)
})
```

Returns: `Promise<PeerInfo[]>` with removed trainer information

`getANRs()`

Get ANR (Autonomous Network Registry) entries.

```
const anrs = await sdk.peer.getANRs()

anrs.forEach((anr) => {
    console.log('Public Key:', anr.pk)
    console.log('IP:', anr.ip4)
    console.log('Port:', anr.port)
    console.log('Version:', anr.version)
})
```

Returns: `Promise<ANRInfo[]>` with ANR information

`getANRByPk(publicKey: string)`

Get ANR entry by public key.

```
const anr = await sdk.peer.getANRByPk('5Kd3N...')

console.log('ANR:', anr)
```

Parameters:

- `publicKey` (string): Public key (Base58 encoded)

Returns: `Promise<ANRInfo>` with ANR information

Proof API

Query validator proofs.

`getValidators(entryHash: string)`

Get validator proof for an entry.

```
const proof = await sdk.proof.getValidators('5Kd3N...')

console.log('Key:', proof.key)
console.log('Value:', proof.value)
console.log('Validators:', proof.validators)
console.log('Proof:', proof.proof)
```

Parameters:

- `entryHash` (string): Entry hash (Base58 encoded)

Returns: `Promise<ProofValidators>` with validator proof

Error Handling

All API methods throw `AmadeusSDKError` on failure:

```
import { AmadeusSDKError } from '@amadeus-protocol/sdk'

try {
    const balance = await sdk.wallet.getBalance(address, 'AMA')
} catch (error) {
    if (error instanceof AmadeusSDKError) {
        console.error('SDK Error:', error.message)
        console.error('Status:', error.status)
        console.error('Response:', error.response)
    }
}
```

Next Steps

- **Utilities**: Learn about utility functions
- **Examples**: See complete API usage examples
- **Best Practices**: Development guidelines

6. Utilities

Complete reference for utility functions in the Amadeus Protocol SDK.

Crypto Utilities

Cryptographic operations for key generation and key derivation.

`generateKeypair()`

Generate a new BLS12-381 keypair.

```
import { generateKeypair } from '@amadeus-protocol/sdk'

const keypair = generateKeypair()

console.log('Public Key:', keypair.publicKey) // Base58 encoded
console.log('Private Key:', keypair.privateKey) // Base58 encoded seed
```

Returns: `KeyPair` with `publicKey` and `privateKey` (both Base58 encoded)

`generatePrivateKey()`

Generate a new random 64-byte seed.

```
import { generatePrivateKey } from '@amadeus-protocol/sdk'

const seed = generatePrivateKey() // Returns Uint8Array (64 bytes)
```

Returns: `Uint8Array` (64 bytes)

`getPublicKey(seed: Uint8Array)`

Derive a public key from a 64-byte seed.

```
import { getPublicKey } from '@amadeus-protocol/sdk'

const seed = generatePrivateKey()
const publicKey = getPublicKey(seed) // Returns Uint8Array (48 bytes)
```

Parameters:

- `seed` (Uint8Array): 64-byte seed

Returns: `Uint8Array` (48 bytes) - BLS12-381 public key

`derivePublicKeyFromSeedBase58(seedBase58: string)`

Derive public key from Base58-encoded seed.

```
import { derivePublicKeyFromSeedBase58 } from '@amadeus-protocol/sdk'

const publicKey = derivePublicKeyFromSeedBase58('5Kd3N...')
console.log('Public Key:', publicKey) // Base58 encoded
```

Parameters:

- `seedBase58` (string): Base58-encoded seed

Returns: `string` - Base58-encoded public key

`deriveSkAndSeed64FromBase58Seed(seedBase58: string)`

Derive secret key and seed from Base58-encoded seed.

```
import { deriveSkAndSeed64FromBase58Seed } from '@amadeus-protocol/sdk'

const { sk, seed64 } = deriveSkAndSeed64FromBase58Seed('5Kd3N...')

console.log('Secret Key:', sk) // Uint8Array
console.log('Seed64:', seed64) // Uint8Array
```

Parameters:

- `seedBase58` (string): Base58-encoded seed

Returns: `{ sk: Uint8Array, seed64: Uint8Array }`

Encoding Utilities

Convert between different binary formats and string encodings.

Base58 Encoding

`toBase58(bytes: Uint8Array): string`

Encode bytes to Base58 string.

```
import { toBase58 } from '@amadeus-protocol/sdk'

const bytes = new Uint8Array([1, 2, 3])
const encoded = toBase58(bytes)
console.log('Base58:', encoded)
```

Parameters:

- `bytes` (Uint8Array): Bytes to encode

Returns: `string` - Base58 encoded string

`fromBase58(str: string): Uint8Array`

Decode Base58 string to bytes.

```
import { fromBase58 } from '@amadeus-protocol/sdk'

const bytes = fromBase58('5Kd3N...')
console.log('Bytes:', bytes)
```

Parameters:

- `str` (string): Base58 string to decode

Returns: `Uint8Array` - Decoded bytes

Base64 Encoding

`uint8ArrayToBase64(bytes: Uint8Array): string`

Convert Uint8Array to Base64 string.

```
import { uint8ArrayToBase64 } from '@amadeus-protocol/sdk'

const bytes = new Uint8Array([1, 2, 3])
const base64 = uint8ArrayToBase64(bytes)
console.log('Base64:', base64)
```

Parameters:

- `bytes` (Uint8Array): Bytes to encode

Returns: `string` - Base64 encoded string

`base64ToUint8Array(base64: string): Uint8Array`

Convert Base64 string to Uint8Array.

```
import { base64ToUint8Array } from '@amadeus-protocol/sdk'

const bytes = base64ToUint8Array('AQID')
console.log('Bytes:', bytes)
```

Parameters:

- `base64` (string): Base64 encoded string

Returns: `Uint8Array` - Decoded bytes

`arrayBufferToBase64(buffer: ArrayBuffer): string`

Convert ArrayBuffer to Base64 string.

```
import { arrayBufferToBase64 } from '@amadeus-protocol/sdk'

const buffer = new ArrayBuffer(8)
const base64 = arrayBufferToBase64(buffer)
```

Parameters:

- `buffer` (ArrayBuffer): Buffer to encode

Returns: `string` - Base64 encoded string

`base64ToArrayBuffer(base64: string): ArrayBuffer`

Convert Base64 string to ArrayBuffer.

```
import { base64ToArrayBuffer } from '@amadeus-protocol/sdk'

const buffer = base64ToArrayBuffer('AQID')
```

Parameters:

- `base64` (string): Base64 encoded string

Returns: `ArrayBuffer` - Decoded buffer

Array Conversion

`uint8ArrayToArrayBuffer(bytes: Uint8Array): ArrayBuffer`

Convert Uint8Array to ArrayBuffer.

```
import { uint8ArrayToArrayBuffer } from '@amadeus-protocol/sdk'

const bytes = new Uint8Array([1, 2, 3])
const buffer = uint8ArrayToArrayBuffer(bytes)
```

Parameters:

- `bytes` (Uint8Array): The Uint8Array to convert

Returns: `ArrayBuffer`

`arrayBufferToUint8Array(buffer: ArrayBuffer): Uint8Array`

Convert ArrayBuffer to Uint8Array.

```
import { arrayBufferToUint8Array } from '@amadeus-protocol/sdk'

const buffer = new ArrayBuffer(8)
const bytes = arrayBufferToUint8Array(buffer)
```

Parameters:

- `buffer` (ArrayBuffer): The ArrayBuffer to convert

Returns: `Uint8Array` - View of the buffer

Serialization

Canonical serialization using VecPack format.

`encode(value: SerializableValue): Uint8Array`

Encode a value using VecPack canonical serialization.

```
import { encode } from '@amadeus-protocol/sdk'

const data = {
    foo: 'bar',
    count: 42,
    items: [1, 2, 3],
    nested: {
        value: true
    }
}

const encoded = encode(data)
console.log('Encoded:', encoded) // Uint8Array
```

Parameters:

- `value`: Serializable value (null, boolean, number, bigint, string, Uint8Array, array, object, Map)

Returns: `Uint8Array` - Encoded bytes

Supported Types:

- `null`
- `boolean`
- `number` / `bigint`
- `string`
- `Uint8Array`
- Arrays
- Objects / Maps

`decode(bytes: Uint8Array | number[]): DecodedValue`

Decode VecPack-encoded bytes.

```
import { decode } from '@amadeus-protocol/sdk'

const decoded = decode(encoded)
console.log('Decoded:', decoded)
```

Parameters:

- `bytes` (Uint8Array | number[]): Encoded bytes

Returns: `DecodedValue` - Decoded value

Conversion Utilities

Convert between human-readable amounts and atomic units.

`toAtomicAma(amount: number): number`

Convert AMA amount to atomic units.

```
import { toAtomicAma } from '@amadeus-protocol/sdk'

const atomic = toAtomicAma(1.5) // Returns 1500000000
console.log('Atomic units:', atomic)
```

Parameters:

- `amount` (number): Amount in AMA (e.g., 1.5)

Returns: `number` - Amount in atomic units

Note: AMA tokens use 9 decimal places.

`fromAtomicAma(atomic: number | string): number`

Convert atomic units to AMA amount.

```
import { fromAtomicAma } from '@amadeus-protocol/sdk'

const ama = fromAtomicAma(1500000000) // Returns 1.5
console.log('AMA:', ama)
```

Parameters:

- `atomic` (number | string): Amount in atomic units

Returns: `number` - Amount in AMA

Encryption Utilities

Password-based encryption for securing sensitive wallet data.

`encryptWithPassword(plaintext: string, password: string): Promise<EncryptedPayload>`

Encrypt plaintext using AES-GCM encryption with PBKDF2 key derivation.

```
import { encryptWithPassword } from '@amadeus-protocol/sdk'

const encrypted = await encryptWithPassword('sensitive data', 'my-
password')

console.log('Encrypted Data:', encrypted.encryptedData) // Base64
console.log('IV:', encrypted.iv) // Base64
console.log('Salt:', encrypted.salt) // Base64

// Store encrypted.encryptedData, encrypted.iv, encrypted.salt
```

Parameters:

- `plaintext` (string): Data to encrypt
- `password` (string): Password for encryption

Returns: `Promise<EncryptedPayload>` - Object containing:

- `encryptedData` (string): Base64-encoded encrypted data
- `iv` (string): Base64-encoded initialization vector
- `salt` (string): Base64-encoded salt

Security Features:

- AES-GCM encryption with 256-bit keys
- PBKDF2 key derivation with 100,000 iterations
- Unique salt and IV for each encryption
- Authenticated encryption (prevents tampering)

`decryptWithPassword(payload: EncryptedPayload, password: string): Promise<string>`

Decrypt encrypted data using the provided password.

```
import { decryptWithPassword } from '@amadeus-protocol/sdk'

const decrypted = await decryptWithPassword(encrypted, 'my-password')
console.log('Decrypted:', decrypted)
```

Parameters:

- `payload` (EncryptedPayload): Encrypted payload with `encryptedData`, `iv`, and `salt`
- `password` (string): Password used for encryption

Returns: `Promise<string>` - Decrypted plaintext

Throws: `Error` if decryption fails (wrong password or corrupted data)

`generateSalt(): Uint8Array`

Generate a cryptographically secure random salt.

```
import { generateSalt } from '@amadeus-protocol/sdk'

const salt = generateSalt() // Returns Uint8Array (16 bytes)
```

Returns: `Uint8Array` - Random salt (16 bytes / 128 bits)

`generateIV(): Uint8Array`

Generate a cryptographically secure random IV for AES-GCM.

```
import { generateIV } from '@amadeus-protocol/sdk'

const iv = generateIV() // Returns Uint8Array (12 bytes)
```

Returns: `Uint8Array` - Random IV (12 bytes / 96 bits)

`deriveKey(password: string, salt: Uint8Array | ArrayBuffer): Promise<CryptoKey>`

Derive an AES-GCM key from a password using PBKDF2.

```
import { deriveKey, generateSalt } from '@amadeus-protocol/sdk'

const salt = generateSalt()
const key = await deriveKey('my-password', salt)
```

Parameters:

- `password` (string): Password string
- `salt` (Uint8Array | ArrayBuffer): Salt for key derivation

Returns: `Promise<CryptoKey>` - Derived AES-GCM key (256 bits)

Constants

`AMADEUS_PUBLIC_KEY_BYTE_LENGTH`

Byte length of BLS12-381 public key: `48`

`AMADEUS_SEED_BYTE_LENGTH`

Byte length of seed: `64`

`AMA_TOKEN_DECIMALS`

Number of decimal places for AMA tokens: `9`

`AMA_TOKEN_DECIMALS_MULTIPLIER`

Multiplier for conversions: `1000000000` (10^9)

`AMA_TRANSFER_FEE`

Network transfer fee: `0.02`

`EXPLORER_URL`

Default explorer URL: `https://explorer.ama.one`

`NODE_API_URL`

Default node API URL: `https://nodes.amadeus.bot/api`

Usage Examples

Complete Encryption Flow

```
import { encryptWithPassword, decryptWithPassword, generateKeypair } from
'@amadeus-protocol/sdk'

// Generate keypair
const keypair = generateKeypair()

// Encrypt private key
const encrypted = await encryptWithPassword(keypair.privateKey, 'user-
password')

// Store encrypted data
localStorage.setItem('encryptedData', encrypted.encryptedData)
localStorage.setItem('iv', encrypted.iv)
localStorage.setItem('salt', encrypted.salt)

// Later: Decrypt private key
const encryptedPayload = {
    encryptedData: localStorage.getItem('encryptedData')!,
    iv: localStorage.getItem('iv')!,
    salt: localStorage.getItem('salt')!
}

const privateKey = await decryptWithPassword(encryptedPayload, 'user-
password')
```

Address Validation

```
import { fromBase58, AMADEUS_PUBLIC_KEY_BYTE_LENGTH } from '@amadeus-
protocol/sdk'

function isValidAddress(address: string): boolean {
    try {
        const bytes = fromBase58(address)
        return bytes.length === AMADEUS_PUBLIC_KEY_BYTE_LENGTH
    } catch {
        return false
    }
}
```

Amount Conversion

```
import { toAtomicAma, fromAtomicAma } from '@amadeus-protocol/sdk'

// Convert for transaction
const amount = 10.5
const atomic = toAtomicAma(amount) // 10500000000

// Convert for display
const display = fromAtomicAma(atomic) // 10.5
```

Next Steps

- **Examples**: See complete utility usage examples
- **Best Practices**: Security and development guidelines
- **API Modules**: Learn about API endpoints

7. Examples

Real-world usage examples for the Amadeus Protocol SDK.

Basic SDK Usage

Initialize SDK and Query Chain

```
import { AmadeusSDK } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK({
    baseUrl: 'https://nodes.amadeus.bot/api'
})

// Get chain tip
const tip = await sdk.chain.getTip()
console.log('Current height:', tip.entry.height)
console.log('Chain hash:', tip.entry.hash)

// Get chain statistics
const stats = await sdk.chain.getStats()
console.log('Total entries:', stats.stats.total_entries)
console.log('Total transactions:', stats.stats.total_transactions)
```

Key Generation

Generate and Use Keypair

```
import { generateKeypair, derivePublicKeyFromSeedBase58 } from '@amadeus-
protocol/sdk'

// Generate new keypair
const keypair = generateKeypair()
console.log('Public Key (Address):', keypair.publicKey)
console.log('Private Key (Seed):', keypair.privateKey) // Keep secret!

// Derive public key from existing seed
const publicKey = derivePublicKeyFromSeedBase58(keypair.privateKey)
console.log('Derived Public Key:', publicKey)
```

Wallet Operations

Check Balance

```
import { AmadeusSDK } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()
const address = '5Kd3N...'

// Get specific token balance
const balance = await sdk.wallet.getBalance(address, 'AMA')
console.log('AMA Balance:', balance.balance.float)

// Get all token balances
const allBalances = await sdk.wallet.getAllBalances(address)
Object.entries(allBalances.balances).forEach(([symbol, balance]) => {
    console.log(`${symbol}: ${balance.float}`)
})
```

Get Transaction History

```
import { AmadeusSDK } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()
const address = '5Kd3N...'

// Get recent transactions
const { txs, cursor } = await sdk.wallet.getTransactions(address, {
    limit: 10,
    sort: 'desc'
})

txs.forEach((tx) => {
    console.log('Hash:', tx.hash)
    console.log('Action:', tx.tx.action)
    console.log('Receipt:', tx.receipt)
})
```

Transaction Building

Simple Token Transfer

```
import { AmadeusSDK, TransactionBuilder, generateKeypair } from
'@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()

// Generate or use existing keypair
const keypair = generateKeypair()
const builder = new TransactionBuilder(keypair.privateKey)

// Build and sign transfer
const { txHash, txPacked } = builder.transfer({
    recipient: '5Kd3N...',
    amount: 10.5,
    symbol: 'AMA'
})

console.log('Transaction hash:', txHash)

// Submit transaction
const result = await sdk.transaction.submit(txPacked)
if (result.error === 'ok') {
    console.log('Transaction submitted:', result.hash)
} else {
    console.error('Error:', result.error)
}
```

Transfer with Confirmation

```
import { AmadeusSDK, TransactionBuilder, generateKeypair } from
'@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()
const keypair = generateKeypair()
const builder = new TransactionBuilder(keypair.privateKey)

// Build and sign
const { txHash, txPacked } = builder.transfer({
    recipient: '5Kd3N...',
    amount: 10.5,
    symbol: 'AMA'
})

// Submit and wait for confirmation
try {
    const result = await sdk.transaction.submitAndWait(txPacked)

    if (result.error === 'ok') {
        console.log('Transaction confirmed!')
        console.log('Hash:', result.hash)
        console.log('Entry hash:', result.metadata?.entry_hash)
        console.log('Receipt:', result.receipt)
    } else {
        console.error('Transaction error:', result.error)
    }
} catch (error) {
    console.error('Transaction failed or timed out:', error)
}
```

Custom Contract Call

```
import { AmadeusSDK, TransactionBuilder, fromBase58, toAtomicAma } from
'@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()
const builder = new TransactionBuilder(privateKey)

// Build custom contract call
const { txHash, txPacked } = builder.buildAndSign('MyContract',
'myMethod', [
    fromBase58('5Kd3N...'),
    toAtomicAma(100).toString(),
    'AMA',
    'additional-arg'
])

const result = await sdk.transaction.submit(txPacked)
```

Build Unsigned, Inspect, Then Sign

```
import { TransactionBuilder } from '@amadeus-protocol/sdk'

const builder = new TransactionBuilder(privateKey)

// Build unsigned transaction
const unsignedTx = builder.buildTransfer({
    recipient: '5Kd3N...',
    amount: 10.5,
    symbol: 'AMA'
})

// Inspect transaction
console.log('Nonce:', unsignedTx.tx.nonce.toString())
console.log('Signer:', unsignedTx.tx.signer)
console.log('Action:', unsignedTx.tx.action)
console.log('Hash:', unsignedTx.hash)

// Sign the transaction
const { txHash, txPacked } = builder.sign(unsignedTx)
```

Contract Interactions

Read Contract Data

```
import { AmadeusSDK, fromBase58 } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()

// Get contract data by key
const key = fromBase58('5Kd3N...')
const data = await sdk.contract.get(key)
console.log('Contract data:', data)

// Get contract data by prefix
const prefixData = await sdk.contract.getPrefix(key)
prefixData.forEach((item) => {
    console.log('Key:', item.key)
    console.log('Value:', item.value)
})
```

Validate Bytecode

```
import { AmadeusSDK } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()

// Load WASM bytecode
const wasmBytecode = await fetch('contract.wasm').then((r) =>
r.arrayBuffer())

// Validate bytecode
const result = await sdk.contract.validateBytecode(wasmBytecode)
if (result.error === 'ok') {
    console.log('Bytecode is valid')
} else {
    console.error('Validation error:', result.error)
}
```

Get Richlist

```
import { AmadeusSDK } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()

const { richlist } = await sdk.contract.getRichlist()

richlist.slice(0, 10).forEach((entry) => {
    console.log(`Rank ${entry.rank}: ${entry.address} - ${entry.balance}
${entry.symbol}`)
})
```

Encryption

Encrypt and Store Private Key

```
import { encryptWithPassword, decryptWithPassword, generateKeypair } from
'@amadeus-protocol/sdk'

// Generate keypair
const keypair = generateKeypair()

// Encrypt private key with user password
const encrypted = await encryptWithPassword(keypair.privateKey, 'user-
password-123')

// Store encrypted data (in real app, use secure storage)
const storage = {
    encryptedData: encrypted.encryptedData,
    iv: encrypted.iv,
    salt: encrypted.salt
}

// Later: Decrypt when user enters password
const decryptedKey = await decryptWithPassword(
    {
        encryptedData: storage.encryptedData,
        iv: storage.iv,
        salt: storage.salt
    },
    'user-password-123'
)

console.log('Decrypted key matches:', decryptedKey ===
keypair.privateKey)
```

Error Handling

Comprehensive Error Handling

```
import { AmadeusSDK, AmadeusSDKError } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()

async function safeGetBalance(address: string, symbol: string) {
    try {
        const balance = await sdk.wallet.getBalance(address, symbol)
        return balance
    } catch (error) {
        if (error instanceof AmadeusSDKError) {
            if (error.status === 404) {
                console.log('Address not found or has no balance')
                return null
            } else if (error.status === 400) {
                console.error('Invalid address format')
                throw new Error('Invalid address')
            } else {
                console.error('SDK Error:', error.message)
                throw error
            }
        } else {
            console.error('Unexpected error:', error)
            throw error
        }
    }
}
```

Complete Transaction Flow

End-to-End Transaction Example

```
import { AmadeusSDK, TransactionBuilder, generateKeypair, toAtomicAma }
from '@amadeus-protocol/sdk'

async function completeTransferFlow() {
    // Initialize SDK
    const sdk = new AmadeusSDK({
        baseUrl: 'https://nodes.amadeus.bot/api'
    })

    // Generate or load keypair
    const keypair = generateKeypair()
    const senderAddress = keypair.publicKey
    const recipientAddress = '5Kd3N...' // Recipient's address

    // Check sender balance
    const balance = await sdk.wallet.getBalance(senderAddress, 'AMA')
    const amount = 10.5
    const transferAmount = toAtomicAma(amount)

    if (balance.balance.flat < transferAmount) {
        throw new Error('Insufficient balance')
    }

    console.log(`Balance: ${balance.balance.float} AMA`)
    console.log(`Transferring: ${amount} AMA`)

    // Build and sign transaction
    const builder = new TransactionBuilder(keypair.privateKey)
    const { txHash, txPacked } = builder.transfer({
        recipient: recipientAddress,
        amount: amount,
        symbol: 'AMA'
    })

    console.log('Transaction hash:', txHash)

    // Submit transaction
    const submitResult = await sdk.transaction.submit(txPacked)

    if (submitResult.error !== 'ok') {
        throw new Error(`Transaction error: ${submitResult.error}`)
    }

    console.log('Transaction submitted:', submitResult.hash)

    // Wait for confirmation
    try {
        const confirmResult = await
sdk.transaction.submitAndWait(txPacked)
```

```
        if (confirmResult.error === 'ok') {
            console.log('Transaction confirmed!')
            console.log('Entry hash:',
confirmResult.metadata?.entry_hash)
            console.log('Receipt:', confirmResult.receipt)
        }
    } catch (error) {
        console.error('Confirmation timeout:', error)
    }

    // Verify transaction
    const tx = await sdk.transaction.get(txHash)
    console.log('Verified transaction:', tx.hash)
}

// Run the flow
completeTransferFlow().catch(console.error)
```

Batch Operations

Submit Multiple Transactions

```
import { AmadeusSDK, TransactionBuilder } from '@amadeus-protocol/sdk'

const sdk = new AmadeusSDK()
const builder = new TransactionBuilder(privateKey)

async function submitBatch(transfers: Array<{ recipient: string; amount:
number }>) {
    const results = []

    for (const transfer of transfers) {
        try {
            // Build and sign
            const { txHash, txPacked } = builder.transfer({
                recipient: transfer.recipient,
                amount: transfer.amount,
                symbol: 'AMA'
            })

            // Submit
            const result = await sdk.transaction.submit(txPacked)

            results.push({
                recipient: transfer.recipient,
                hash: result.hash,
                error: result.error
            })

            // Small delay to avoid nonce collisions
            await new Promise((resolve) => setTimeout(resolve, 100))
        } catch (error) {
            results.push({
                recipient: transfer.recipient,
                error: error.message
            })
        }
    }

    return results
}
```

Next Steps

- **Best Practices**: Learn security and development guidelines
- **Troubleshooting**: Common issues and solutions
- **API Modules**: Complete API reference

8. Best Practices

Security and development best practices for using the Amadeus Protocol SDK.

Security

Private Key Management

Never:

- ❌ Commit private keys to version control
- ❌ Log private keys in console or files
- ❌ Share private keys with anyone
- ❌ Store private keys in plain text
- ❌ Send private keys over unencrypted channels

Always:

- ✅ Use environment variables for private keys
- ✅ Encrypt private keys at rest
- ✅ Use secure key storage solutions
- ✅ Use separate keys for development and production
- ✅ Implement proper access controls

```
// ✅ Good: Use environment variables
const privateKey = process.env.PRIVATE_KEY
if (!privateKey) {
    throw new Error('PRIVATE_KEY not set')
}

// ❌ Bad: Hardcoded private key
const privateKey = '5Kd3N...' // DON'T DO THIS!
```

Password-Based Encryption

Always encrypt sensitive data before storage:

```
import { encryptWithPassword, decryptWithPassword } from '@amadeus-
protocol/sdk'

// Encrypt before storage
const encrypted = await encryptWithPassword(privateKey, userPassword)

// Store encrypted data securely
await secureStorage.save({
    encryptedData: encrypted.encryptedData,
    iv: encrypted.iv,
    salt: encrypted.salt
})

// Decrypt only when needed
const decrypted = await decryptWithPassword(encrypted, userPassword)
```

Address Validation

Always validate addresses before using them:

```
import { fromBase58, AMADEUS_PUBLIC_KEY_BYTE_LENGTH } from '@amadeus-
protocol/sdk'

function isValidAddress(address: string): boolean {
    try {
        const bytes = fromBase58(address)
        return bytes.length === AMADEUS_PUBLIC_KEY_BYTE_LENGTH
    } catch {
        return false
    }
}

// Use validation
if (!isValidAddress(recipient)) {
    throw new Error('Invalid recipient address')
}
```

Transaction Verification

Always verify transaction details before signing:

```
// Build unsigned transaction first
const unsignedTx = builder.buildTransfer({
    recipient: address,
    amount: amount,
    symbol: 'AMA'
})

// Verify details
console.log('Recipient:', toBase58(unsignedTx.tx.action.args[0]))
console.log('Amount:', unsignedTx.tx.action.args[1])
console.log('Symbol:', unsignedTx.tx.action.args[2])

// Only sign after verification
const { txHash, txPacked } = builder.sign(unsignedTx)
```

Error Handling

Comprehensive Error Handling

Always handle errors appropriately:

```
import { AmadeusSDKError } from '@amadeus-protocol/sdk'

try {
    const result = await sdk.transaction.submit(txPacked)

    if (result.error === 'ok') {
        console.log('Success:', result.hash)
    } else {
        // Handle specific transaction errors
        switch (result.error) {
            case 'insufficient_funds':
                throw new Error('Not enough balance')
            case 'invalid_signature':
                throw new Error('Invalid signature')
            default:
                throw new Error(`Transaction error: ${result.error}`)
        }
    }
} catch (error) {
    if (error instanceof AmadeusSDKError) {
        // Handle SDK-specific errors
        if (error.status === 404) {
            console.log('Resource not found')
        } else if (error.status === 400) {
            console.error('Invalid request:', error.message)
        } else {
            console.error('SDK Error:', error.message)
        }
    } else {
        // Handle unexpected errors
        console.error('Unexpected error:', error)
    }
}
```

Retry Logic

Implement retry logic for network requests:

```
async function submitWithRetry(
    txPacked: Uint8Array,
    maxRetries = 3,
    delay = 1000
): Promise<SubmitTransactionResponse> {
    for (let i = 0; i < maxRetries; i++) {
        try {
            return await sdk.transaction.submit(txPacked)
        } catch (error) {
            if (i === maxRetries - 1) throw error

            // Exponential backoff
            await new Promise((resolve) => setTimeout(resolve, delay *
Math.pow(2, i)))
        }
    }
    throw new Error('Max retries exceeded')
}
```

Transaction Management

Nonce Management

For high-frequency transactions, ensure sufficient time between transactions:

```
async function submitMultipleTransactions(txs: Uint8Array[]) {
    for (const tx of txs) {
        await sdk.transaction.submit(tx)

        // Add delay to avoid nonce collisions
        await new Promise((resolve) => setTimeout(resolve, 100))
    }
}
```

Balance Checking

Always check balance before transferring:

```
async function safeTransfer(recipient: string, amount: number, symbol:
string) {
    // Check balance first
    const balance = await sdk.wallet.getBalance(senderAddress, symbol)
    const transferAmount = toAtomicAma(amount)

    if (balance.balance.flat < transferAmount) {
        throw new Error('Insufficient balance')
    }

    // Proceed with transfer
    const { txHash, txPacked } = builder.transfer({
        recipient,
        amount,
        symbol
    })

    return await sdk.transaction.submit(txPacked)
}
```

Amount Precision

Always use conversion functions for amounts:

```
import { toAtomicAma, fromAtomicAma } from '@amadeus-protocol/sdk'

// ✅ Good - use conversion function
const amount = toAtomicAma(1.5)

// ❌ Bad - may lose precision
const amount = 1.5 * 1000000000
```

Configuration

Environment-Based Configuration

Use environment variables for configuration:

```
const sdk = new AmadeusSDK({
    baseUrl: process.env.NODE_API_URL || 'https://nodes.amadeus.bot/api',
    timeout: parseInt(process.env.REQUEST_TIMEOUT || '30000')
})
```

Request Timeouts

Set appropriate timeouts for your use case:

```
// Short timeout for quick queries
const quickSDK = new AmadeusSDK({
    baseUrl: 'https://nodes.amadeus.bot/api',
    timeout: 5000 // 5 seconds
})

// Longer timeout for transactions
const txSDK = new AmadeusSDK({
    baseUrl: 'https://nodes.amadeus.bot/api',
    timeout: 60000 // 60 seconds
})
```

Code Organization

Separate Concerns

Organize your code into logical modules:

```
// wallet.ts
export class WalletManager {
    constructor(
        private sdk: AmadeusSDK,
        private builder: TransactionBuilder
    ) {}

    async getBalance(address: string, symbol: string) {
        return this.sdk.wallet.getBalance(address, symbol)
    }

    async transfer(recipient: string, amount: number, symbol: string) {
        const { txHash, txPacked } = this.builder.transfer({
            recipient,
            amount,
            symbol
        })
        return this.sdk.transaction.submit(txPacked)
    }
}

// chain.ts
export class ChainQuerier {
    constructor(private sdk: AmadeusSDK) {}

    async getCurrentHeight() {
        const tip = await this.sdk.chain.getTip()
        return tip.entry.height
    }

    async getStats() {
        return this.sdk.chain.getStats()
    }
}
```

Type Safety

Use TypeScript types for better safety:

```
import type { AmadeusSDKConfig, Transaction, WalletBalance } from
'@amadeus-protocol/sdk'

function processTransaction(tx: Transaction) {
    // TypeScript will catch type errors
    console.log(tx.hash)
    console.log(tx.tx.action)
}
```

Testing

Testnet Usage

Always test on Testnet first:

```
// Testnet configuration
const testnetSDK = new AmadeusSDK({
    baseUrl: 'https://testnet-rpc.ama.one/api'
})

// Test transactions on testnet
const testResult = await testnetSDK.transaction.submit(testTxPacked)
```

Mock Implementations

Create mocks for testing:

```
class MockAmadeusSDK {
  async wallet = {
    getBalance: async () => ({
      balance: { float: 100, flat: 100000000000, symbol: 'AMA' }
    })
  }

  async transaction = {
    submit: async () => ({ error: 'ok', hash: 'mock-hash' })
  }
}
```

Performance

Batch Operations

Batch operations when possible:

```
// ✅ Good: Batch balance queries
const addresses = ['addr1', 'addr2', 'addr3']
const balances = await Promise.all(addresses.map((addr) =>
sdk.wallet.getBalance(addr, 'AMA')))

// ❌ Bad: Sequential queries
for (const addr of addresses) {
    await sdk.wallet.getBalance(addr, 'AMA')
}
```

Caching

Cache frequently accessed data:

```
class CachedChainQuerier {
    private tipCache: ChainEntry | null = null
    private cacheTime = 0
    private cacheTTL = 5000 // 5 seconds

    async getTip(): Promise<ChainEntry> {
        const now = Date.now()
        if (this.tipCache && now - this.cacheTime < this.cacheTTL) {
            return this.tipCache
        }

        const { entry } = await this.sdk.chain.getTip()
        this.tipCache = entry
        this.cacheTime = now
        return entry
    }
}
```

Logging

Structured Logging

Use structured logging:

```
function logTransaction(txHash: string, result:
SubmitTransactionResponse) {
    console.log(
        JSON.stringify({
            type: 'transaction',
            hash: txHash,
            error: result.error,
            timestamp: new Date().toISOString()
        })
    )
}
```

Sensitive Data

Never log sensitive data:

```
// ✅ Good: Log only public information
console.log('Transaction hash:', txHash)
console.log('Recipient:', recipientAddress)

// ❌ Bad: Log private keys
console.log('Private key:', privateKey) // NEVER DO THIS!
```

Documentation

Code Comments

Document complex logic:

```
/**
 * Builds and submits a transfer transaction with balance verification
 *
 * @param recipient - Base58 encoded recipient address
 * @param amount - Amount in AMA (human-readable)
 * @param symbol - Token symbol (default: 'AMA')
 * @returns Transaction hash if successful
 * @throws Error if balance is insufficient or transaction fails
 */
async function transferWithVerification(
    recipient: string,
    amount: number,
    symbol: string = 'AMA'
): Promise<string> {
    // Implementation...
}
```

Next Steps

- **Troubleshooting**: Common issues and solutions
- **Examples**: Real-world usage examples
- **API Modules**: Complete API reference

9. Troubleshooting

Common issues and solutions when using the Amadeus Protocol SDK.

Installation Issues

"Module not found" Error

Problem: Cannot find module '@amadeus-protocol/sdk'

Solutions:

1. Ensure the package is installed:

```
npm install @amadeus-protocol/sdk
```

2. Check your `package.json`:

```
{
    "dependencies": {
        "@amadeus-protocol/sdk": "^1.0.0"
    }
}
```

3. Clear cache and reinstall:

```
npm cache clean --force
rm -rf node_modules package-lock.json
npm install
```

TypeScript Errors

Problem: Type errors when importing

Solutions:

1. Ensure TypeScript is installed:

```
npm install -D typescript @types/node
```

2. Check `tsconfig.json`:

```
{
    "compilerOptions": {
        "moduleResolution": "node",
        "esModuleInterop": true
    }
}
```

3. Restart your TypeScript server/IDE

Node.js Version Issues

Problem: Errors related to Node.js version

Solutions:

1. Check Node.js version:

```
node --version
```

2. Update to Node.js 20+ if needed
3. Use a version manager like `nvm`:

```
nvm install 20
nvm use 20
```

Runtime Errors

"Invalid Base58 string"

Problem: Error when decoding Base58 addresses or keys

Causes:

- Invalid Base58 encoding
- Missing or extra characters
- Wrong string format

Solutions:

```
import { fromBase58 } from '@amadeus-protocol/sdk'

function validateBase58(str: string): boolean {
    try {
        fromBase58(str)
        return true
    } catch {
        return false
    }
}

// Validate before use
if (!validateBase58(address)) {
    throw new Error('Invalid Base58 address')
}
```

"Transaction failed: insufficient balance"

Problem: Transaction rejected due to insufficient funds

Solutions:

1. Check balance before transferring:

   ```
   const balance = await sdk.wallet.getBalance(address, 'AMA')
   if (balance.balance.float < amount + fee) {
       throw new Error('Insufficient balance')
   }
   ```

2. Account for transaction fees:

   ```
   const requiredAmount = amount + AMA_TRANSFER_FEE
   ```

"Request timeout"

Problem: API requests timing out

Causes:

- Network connectivity issues
- Node is slow or overloaded
- Timeout too short

Solutions:

1. Increase timeout:

```
const sdk = new AmadeusSDK({
    baseUrl: 'https://nodes.amadeus.bot/api',
    timeout: 60000 // 60 seconds
})
```

2. Implement retry logic:

```
async function submitWithRetry(txPacked: Uint8Array, maxRetries = 3) {
    for (let i = 0; i < maxRetries; i++) {
        try {
            return await sdk.transaction.submit(txPacked)
        } catch (error) {
            if (i === maxRetries - 1) throw error
            await new Promise((resolve) => setTimeout(resolve, 1000 *
(i + 1)))
        }
    }
}
```

"Invalid transaction structure"

Problem: Transaction validation fails

Causes:

- Transaction not properly built
- Invalid arguments
- Wrong contract/function names

Solutions:

1. Always use `TransactionBuilder`:

```
const builder = new TransactionBuilder(privateKey)
const { txPacked } = builder.transfer({
    recipient: address,
    amount: amount,
    symbol: 'AMA'
})
```

2. Validate inputs:

```
function validateTransferInput(input: TransferInput): boolean {
    return (
        isValidAddress(input.recipient) && input.amount > 0 && typeof
input.symbol === 'string'
    )
}
```

"Invalid signature"

Problem: Transaction signature validation fails

Causes:

- Wrong private key used
- Transaction modified after signing
- Corrupted transaction data

Solutions:

1. Verify private key matches address:

```
const publicKey = derivePublicKeyFromSeedBase58(privateKey)
if (publicKey !== expectedAddress) {
    throw new Error('Private key does not match address')
}
```

2. Don't modify transactions after signing:

```
// ✅ Good: Use txPacked directly
await sdk.transaction.submit(txPacked)

// ❌ Bad: Don't modify after signing
// const modified = modifyTransaction(txPacked)
```

API Errors

404 Not Found

Problem: Resource not found

Solutions:

```
try {
    const balance = await sdk.wallet.getBalance(address, 'AMA')
} catch (error) {
    if (error instanceof AmadeusSDKError && error.status === 404) {
        console.log('Address not found or has no balance')
        // Handle gracefully
    }
}
```

400 Bad Request

Problem: Invalid request parameters

Solutions:

1. Validate inputs before API calls:

```
function isValidAddress(address: string): boolean {
    try {
        const bytes = fromBase58(address)
        return bytes.length === 48
    } catch {
        return false
    }
}
```

2. Check error response for details:

```
catch (error) {
  if (error instanceof AmadeusSDKError) {
    console.error('Error details:', error.response)
  }
}
```

Network Errors

Problem: Network connectivity issues

Solutions:

1. Check network connectivity:

```
async function checkConnection(): Promise<boolean> {
    try {
        await sdk.chain.getTip()
        return true
    } catch {
        return false
    }
}
```

2. Use different node URL:

```
const sdk = new AmadeusSDK({
    baseUrl: 'https://backup-node.com/api'
})
```

Transaction Issues

Transaction Stuck/Pending

Problem: Transaction submitted but not confirming

Solutions:

1. Check transaction status:

```
const tx = await sdk.transaction.get(txHash)
console.log('Transaction status:', tx.receipt)
```

2. Wait for confirmation:

```
try {
    const result = await sdk.transaction.submitAndWait(txPacked)
    console.log('Confirmed:', result.metadata?.entry_hash)
} catch (error) {
    console.error('Confirmation timeout')
}
```

Nonce Collisions

Problem: Multiple transactions with same nonce

Causes:

- Transactions submitted too quickly
- Timestamp-based nonce collision

Solutions:

1. Add delays between transactions:

```
async function submitMultiple(txs: Uint8Array[]) {
    for (const tx of txs) {
        await sdk.transaction.submit(tx)
        await new Promise((resolve) => setTimeout(resolve, 100))
    }
}
```

Wrong Amount

Problem: Amount not matching expected value

Causes:

- Precision loss from manual calculation
- Wrong conversion

Solutions:

1. Always use conversion functions:

```
// ✅ Good
const atomic = toAtomicAma(1.5)

// ❌ Bad
const atomic = 1.5 * 1000000000
```

2. Verify amounts:

```
const amount = 10.5
const atomic = toAtomicAma(amount)
const back = fromAtomicAma(atomic)
console.log('Matches:', amount === back) // Should be true
```

Encryption Issues

"Decryption failed"

Problem: Cannot decrypt encrypted data

Causes:

- Wrong password
- Corrupted encrypted data
- Missing IV or salt

Solutions:

1. Verify password:

```
try {
    const decrypted = await decryptWithPassword(encrypted, password)
} catch (error) {
    if (error.message.includes('decryption')) {
        console.error('Wrong password or corrupted data')
    }
}
```

2. Ensure all components are present:

```
const encrypted = {
    encryptedData: stored.encryptedData,
    iv: stored.iv,
    salt: stored.salt
}
```

Debugging

Enable Verbose Logging

```
// Log all API requests
const originalGet = sdk.client.get.bind(sdk.client)
sdk.client.get = async (...args) => {
  console.log('GET:', args)
  return originalGet(...args)
}

// Log transaction building
const builder = new TransactionBuilder(privateKey)
const unsignedTx = builder.buildTransfer({...})
console.log('Unsigned transaction:', {
  nonce: unsignedTx.tx.nonce.toString(),
  action: unsignedTx.tx.action,
  hash: toBase58(unsignedTx.hash)
})
```

Transaction Inspection

```
// Inspect transaction before signing
const unsignedTx = builder.buildTransfer({
    recipient: address,
    amount: amount,
    symbol: 'AMA'
})

console.log('Transaction details:', {
    signer: toBase58(unsignedTx.tx.signer),
    nonce: unsignedTx.tx.nonce.toString(),
    contract: unsignedTx.tx.action.contract,
    function: unsignedTx.tx.action.function,
    args: unsignedTx.tx.action.args,
    hash: toBase58(unsignedTx.hash)
})
```

Getting Help

Check Documentation

1. Review API Reference
2. Check Examples
3. Read Best Practices

Common Resources

- **GitHub Issues**: Report bugs and request features
- **Documentation**: Complete API reference
- **Examples**: Real-world usage examples

Reporting Issues

When reporting issues, include:

1. SDK version
2. Node.js version
3. Error message and stack trace
4. Code snippet that reproduces the issue
5. Expected vs actual behavior

Next Steps

- **Best Practices**: Security and development guidelines
- **Examples**: Real-world usage examples
- **API Modules**: Complete API reference

TESTNET

Connect to Testnet0

Testnet0 is available on:

- TESTNET RPC https://testnet.ama.one/
- TESTNET testnet.ama.one
- TESTNET 46.4.179.184

Remember: Do not use your mainnet keys to interact with the testnet.

To connect the explorer or wallet on Testnet0 you need to redirect the `nodes.amadeus.bot` RPC to point to the testnet RPC. The easiest way to do this is via hosts file by adding an extra line

```
vim /etc/hosts
46.4.179.184 nodes.amadeus.bot
```

Another way is via host resolver in chrome

```
--host-resolver-rules="MAP nodes.amadeus.bot testnet.ama.one"
```

Next your browser when opening the explorer or wallet will give an SSL error so open a new Chrome profile dropping all security checks:

```
mkdir -p /tmp/chrome_testnet0

google-chrome  --user-data-dir="/tmp/chrome_testnet0" \
--no-first-run --no-default-browser-check \
--ignore-certificate-errors --disable-web-security \
--unsafely-treat-insecure-origin-as-secure=https://nodes.amadeus.bot

[OPTIONAL] if you did not use hosts file
--host-resolver-rules="MAP nodes.amadeus.bot testnet.ama.one"
```

Now you can open the following URLs in that isolated browser profile and all will work

- https://ama-explorer.ddns.net/

- https://wallet.ama.one/

Faucet for Testnet0

For now ask in Discord for coins

OR

solve MatMul to get emissions from epoch rewards.

Local Testnet

How to create a wallet

Remember: Do not use your mainnet keys to interact with the testnet.

The quickest way to build and debug without deploying contracts over and over is to test locally.

A testnet can be run fully local to your PC with validators and block production, mimicking real timings, finality and more.

Local testnet spawns with 10 validators that each have 10m AMA.

Follow the `Connect to Testnet0` guide first, but replace the redirected host to `localhost` or `127.0.0.1`

This way things like the RPC or Wallet now will use your locally running testnet node as the source for the RPC.

You might run into some problems so lets just outline how to run the node first, one thing if you are not root is you wont be allowed to listen on ports below 1024, to fix this, grant permission to bind port 80 and 443 for RPC.

```
#allow listening on port 80 and 443
sudo sysctl -w net.ipv4.ip_unprivileged_port_start=80
```

Now run the node via `amadeusd`

```
#run the local testnet
TESTNET=true WORKFOLDER=/tmp/testnet HTTP_IPV4=127.0.0.1 HTTP_PORT=80 ./
amadeusd
```

Confirm its working by submitting a transaction between validators using the `REPL` `iex(1)>`

```
key0 = Application.fetch_env!(:ama, :keys) |> Enum.at(0)
key1 = Application.fetch_env!(:ama, :keys) |> Enum.at(1)
Testnet.call(key0.seed, "Coin", "transfer", [key1.pk, "1", "AMA"])
```

Result

```
%{
  entry_hash: "7ELicUfkqo69cdJcTsbinpKyQEbuS5raAHgSHmVknYPk",
  error: :ok,
  hash: "DhzK2FGMsSLFiqHujSzkeo1s7nKKqqWAHnydRShMPn5G",
  result: %{error: "ok", exec_used: "10000000"}
}
```

CLI

Using the REPL

The (R)ead (E)valuate (P)rint (L)oop is a core feature of the CLI node. It allows you to execute direct commands inside a running node without having to code everything into a command or API.

For example you can query the status of the network by executing `API.Chain.stats()` inside the REPL of a running and online node.

```
iex(114)> API.Chain.stats()
%{
  height: 29984500,
  pflops: 27.869284354182614,
  circulating: 330546993.68658775,
  cur_validator:
"6V65RDdHU8T7TbFxGh42sp2hmXrfmRRFbuTjmJv4yysikdNhtdSC2yMxr7L95gDCKn",
  emission_for_epoch: 905214.287460235,
  next_validator:
"7pGZFZpfJbUf8NwqSw84cRvNGabvDdd2SafxvNdTUh7Peus2S1tBv21ETiq46Xg4kb",
  tip: %{
    hash: "AxUjrFBJQZ4kUac42S6qoELG3iHFwFK4aKwrFVYFfXtU",
    tx_count: 1,
    header_unpacked: %{
      slot: 29984500,
      dr: "3UaAR71UXvApB3qCvt6dwwrdgERWKhgcmS68yjSLcKjc",
      vr:
"22pssGv5SwdJxxD4Q4UwN4upmwQ42ivb8MkY44AW7JczfMTKofjbNirsmAZ6ztAtqdfi1c4w
4GZ1MHanqhoRptcKAhU5oTT9P5zboPQ5QKj8sPRigCLmqFAD28cMFMLsDWQT",
      prev_hash: "8VHwdFDXYS2LHukKJYa4rxsrTwyrswJ9mYcwtpMoaCom",
      signer:
"6V65RDdHU8T7TbFxGh42sp2hmXrfmRRFbuTjmJv4yysikdNhtdSC2yMxr7L95gDCKn",
      height: 29984500,
      prev_slot: 29984499
    },
    consensus: %{score: 0.848, mut_hash:
"4GB4rBv2GGXaQ7TCYPV7THsFzov1KnTkomedKjMW4LN9"}
  },
  tip_hash: "AxUjrFBJQZ4kUac42S6qoELG3iHFwFK4aKwrFVYFfXtU",
  tx_pool_size: 0
}
```

The future instructions here will all make use of the REPL from an OFFLINE running node. An OFFLINE node does not join the network but can communicate with the RPC API of other nodes.

To run a node in offline mode download the latest `amadeusd` release from github https://github.com/amadeus-robot/node/releases/latest and run it like:

```
OFFLINE=true ./amadeusd
```

Creating a Wallet

Key pairs created by the CLI are offline.

You are responsible for safe storage of your seed phrase.

```
{public_key, seed} = API.Wallet.generate_key()
```

```
{"6szjTAfFU1KqxGVqyhuhGoA9GTJH93dEw6hDccVur2KV3Tc3juFfW5vYTENFe3hcPc",

"2k3LidUYf6c6Lxs5YR2RY9mT7gJS6663XbyfarMQx8V5Pf28NF5XCQaecZrvYty6sKHxr5w4
RoUct2t16agNTAQd"}
```

The output is your Base58 encoded BLS12-381 - 48byte public key and 64 byte seed.

Your receiver wallet address is your public key.

Keep your seed secret.

Broadcast Transaction

Transaction API requires just your seed.

To create a new transfer funds transaction from the wallet we created earlier, send the funds to the donation address and broadcast it to the network run the command

```
seed64 =
"2k3LidUYf6c6Lxs5YR2RY9mT7gJS6663XbyfarMQx8V5Pf28NF5XCQaecZrvYty6sKHxr5w4
RoUct2t16agNTAQd"
receiver =
"69TDon8KJp3vicNeFR3dg5x5sKY8PJFLmoizX3RN31YL4fwr266AVcXgwy1mjCLy6M"
amount = 1.0  # AMA has 9 decimals. the API converts type float to
integer currency
              # 1.0 is actually 1_000_000_000 AMA
              # if you pass amount = 1 you will send 1 of the lowest
denomination of AMA
symbol = "AMA"

RPC.API.Wallet.transfer(seed64, receiver, amount, symbol)
%{error: "ok", hash: "CNzM2qvuPBVgV92LnBDtSrbFAm3iTwupEWiVGEBmrKpy"}
#Wait a second
RPC.API.Wallet.balance(receiver, symbol)
RPC.API.Chain.tx("CNzM2qvuPBVgV92LnBDtSrbFAm3iTwupEWiVGEBmrKpy")
```

Your transaction will be broadcast to an RPC node and added to the TXPool.

Reasons your TX might not be found are mainly it failed due to not having enough balance on your account, or you sent all the coins off the address and did not leave enough to pay for exec costs (always leave at least 0.1 AMA). In the future there will be dust management.

Sending OTHER coins

If you wish to send something other than AMA you can specify via the 4th argument the symbol name:

```
RPC.API.Wallet.transfer(seed64, receiver, amount, "NEURAL")
```

Bulk Transfer

Size 2 tuple implies AMA, otherwise 3rd element is the symbol to send.

```
seed64 =
"2k3LidUYf6c6Lxs5YR2RY9mT7gJS6663XbyfarMQx8V5Pf28NF5XCQaecZrvYty6sKHxr5w4
RoUct2t16agNTAQd"
RPC.API.Wallet.transfer_bulk(seed64, [
  {"7dEiLC5JQZKYz8HENiqVF4nT1HouyTsii213TXPrYySoKYXUuecUWgnnnux7hJty19",
1.0},
  {"7EDiQVxyKQjf6sKSYTeAkuK5vDwXHcvno2kZeP8VJW8gZcUuLV9RJ54vveTiKP4a36",
1_000_000_000},
  {"78pX5oy35LeJPkfP6A4nKKtvSfynzAEoCPvHAQgSpwLgcADgt5Vd9vESSqre3yQHm1",
1.0, "USDT"},
])
```

Validator

Running a Node

The basic requirements in order of importance:

- 1gbps+ stable internet connection
- modern CPU with AVX512 (7950x is best value for $)
- 2TB+ diskspace
- UDP port 36969 open

The basic system configuration required anything less will degrade performance:

```
/etc/sysctl.conf
```

```
net.core.wmem_max = 268435456
net.core.rmem_default = 212992
net.core.rmem_max = 268435456
net.core.netdev_max_backlog = 300000
net.core.optmem_max = 268435456
net.ipv4.udp_mem = 3060432 4080578 6120864

net.ipv4.conf.all.rp_filter=1
net.ipv4.conf.default.rp_filter=1
```

```
/etc/security/limits.conf
```

```
root hard nofile 1048576
root soft nofile 1048576
* hard nofile 1048576
* soft nofile 1048576
root hard nproc unlimited
root soft nproc unlimited
* hard nproc unlimited
* soft nproc unlimited
root hard memlock unlimited
root soft memlock unlimited
* hard memlock unlimited
* soft memlock unlimited
```

To run a node download the latest `amadeusd` release from github https://github.com/amadeus-robot/node/releases/latest and run it like:

```
./amadeusd
```

Multiple .envvars are supported. For the best up-to-date list check https://github.com/amadeus-robot/node/blob/main/ex/config/runtime.exs

Here are a few important ones with their respective defaults:

```
WORKFOLDER=~/.cache/amadeusd/    # where the blockchain + all data is
stored
OFFLINE=false                    # run the node without connecting to any
peers

UDP_IPV4=0.0.0.0                 # ip address(es) to listen on for
protocol.
                                 # default listen on all interfaces
UDP_PORT=36969                   # mainnet port

PUBLIC_UDP_IPV4=stun             # if behind complex NAT, what IP do you
want to advertise
                                 # default - obtain ip via STUN

ANR_NAME=nil                     # (optional) name of your validator
ANR_DESC=nil                     # (optional) description of your
validator
                                 # stored in https://nodes.amadeus.bot/
api/peer/anr

HTTP_IPV4=off                    # HTTP RPC API (useful for integrations)
HTTP_PORT=80

ARCHIVALNODE=false               # if this node is expected to store all
chainstate
                                 # for integrations enable this

COMPUTOR=false                   # if this node will also solve (for
mainnet disable)
```

Your seed for your public key is stored in `$WORKFOLDER/sk`.

Syncing currently takes a long time as the entire chainstate is pulled including contract states + transactions since genesis. This is just over 170G compressed as of September 19th, 2025. Do not attempt syncing unless you have stable 1gbps connection.

Here is a systemd service that will restart the node as required and run it on a `screen`:

```
/etc/systemd/system/amadeusd.service
```

```
[Unit]
Description=AmadeusD
After=network-online.target

[Service]
Type=forking
LimitNOFILE=1048576
KillMode=control-group
Restart=always
RestartSec=3
User=root
WorkingDirectory=/root/
Environment="UDP_IPV4=0.0.0.0"
ExecStart=/usr/bin/screen -UdmS amadeusd bash -c './amadeusd'

[Install]
WantedBy=default.target
```

Building your Score

The top 32 (as of September 19, 2025) node scores of each epoch enter into the validator list for the next epoch. Validators that are already in the current epoch and make it to the next epoch receive emissions.

Validators who made top 32 but drop out in next epoch receive no emissions, we are currently thinking about how to make this more forgiving to encourage adoption and lower validator risk. If you have suggestions please share.

To build your score you need to produce solutions that are seeded with your validators key. The current protocol level parse of the seed is here https://github.com/amadeus-robot/node/blob/main/ex/lib/bic/sol.ex and as of September 19, 2025 its built like so:

```
<<
  epoch::32-little,
  segment_vr_hash::32-binary,
  node_pk::48-binary,
  node_pop::96-binary,
  solver_pk::48-binary,         # (optional) hint who solved the sol
  nonce::12-binary
>>

seed = <<
  Consensus.chain_epoch()::32-little,
  Consensus.chain_segment_vr_hash()::32-binary,
  Application.fetch_env!(:ama, :trainer_pk)::48-binary,
  Application.fetch_env!(:ama, :trainer_pop)::96-binary,
  Application.fetch_env!(:ama, :trainer_pk)::48-binary,
  :crypto.strong_rand_bytes(12)::12-binary
>>

b = Blake3.new()
Blake3.update(b, seed)
<<
   matrix_a::binary-size(16*50240),
   matrix_b::binary-size(50240*16)
>> = Blake3.finalize_xof(b, 16*50240 + 50240*16)
c = MatrixMul.multiply(matrix_a, matrix_b) |> MatrixMul.map_to_binary()
solution = seed <> c

diff_bits = Consensus.chain_diff_bits()
<<leading_zeros::size(diff_bits), _::bitstring>> = hash =
Blake3.hash(solution)
if leading_zeros == 0 do
  IO.puts "congratulations, broadcast this now to the network to increase
score"
  tx_packed = TX.build(Application.fetch_env!(:ama, :trainer_sk),
    "Epoch", "submit_sol", [solution])
  TXPool.insert_and_broadcast(tx_packed)
end
```

Risks of Validating

Once the new epoch starts and your node score is in the top 32:

```
API.Epoch.score()

[
  ["7DXyX5siLBPbkwhW2aaV4WRuLAHu7LTn2BJFNvprZ5wt9jtrXXusrys7qKmJdBhx1n",
82307],
  ["6u6Ym7mx3aqajGZHqdYP4eF8pNrFHx9vCEWNRa71ijTQhxSqTnjsR8RJc4DYTYUuxo",
9504],
  ["5zU13Q4iNrcxYenX5oUU8h7YAzM7XqQAzMe9fHs9dyxf58HMnA4JMX9vC7q5GiGpKa",
9493],
  ["6L9Tuur76iEd862SoERbuhHFFjWh2V5kMLAZhdKL5aYiQivCNfYYniPxogu4yLwpQ9",
9178],
  ["6bntFSNoGRQtaJkqD9czp9RxkiajVYg9mdnYd6h8gV3CFZKUd4eC6vCGWWfXHjMUXX",
9098],
  ["6V7zuE9Ci4RHWVDZxrKDK6sKR6QMEHdW7BrwadTR5N8dqyvE1VYEEGq9gWapPdy461",
9042],
  ["71botgABuPbNq8rfTtcEB4yrpo8SfLP22eos6cHTr2okmraU8TrFk7yzSBMhFfhRnq",
8997],
  ["6AK24JHCvsSvrVd9e1nntUd6ViKKq96cfnQpfvSDYivxt6MFvsGMb6DHRkQmP3huZU",
8783],
  ["7eQUenPHLRKvYGgCFS5CU1Si6qa4HnQps4uxXiQo587xu8aBHRXZRFpUPzNSAM2Fdj",
8693],
  ["667rUL2QG8EjedXzrmuHuBVoACQqmYgGxi65KR7JwJE1m9DXMXtrZ9hwSqcawvZrXo",
8598],
  ["7pGZFZpfJbUf8NwqSw84cRvNGabvDdd2SafxvNdTUh7Peus2S1tBv21ETiq46Xg4kb",
8338],
  ["6Z1xwWG5fNudY81gfL695DfJWh3uhmGvNsv2Q6aUotZudUHPqLeAjCxeDt5WXj7MBz",
8316],
  ["7VgGc6ZfALjrRyK6fxQeV6gZeeGpByoPx1uU5gdb1y1DVgLxyBFSa9vSQQFVhiprEp",
8202],
  ["6H3pRBo5snb5qNaehR9DBcm57YCvPaBDED5sRQ1LT7oSZuBpnib7VdsduHR2ojweFz",
8137],
  ["5qYZD8HVJzXX4UJDCe6gHdAu4or2jWdnkUX1ZLdGxBx3cKj6NkbXNz46CGxMjqyuL1",
8088],
  ["6ddsvp1auJ2zWqDwFJdoHvijEsHyXLfS2yFnxmyCBfgU7PsNRoeMgJSRhdforShBk5",
8077],
  ["7SswepNgUigEMiaxtVPHCU8pCJjNXAqp1ZZgEaqZGAtZZThqmg9JnjPPJ4fzJTHhHY",
8070],
  ["7nZ1tCQtfMBeFeJ8DGYX55QkWB7CtNpDfKgoehXrmUetVN68bAPNSzLYU76Bkz8kDv",
8025],
  ["7WH7ZHYwQZ6Fq3E95gzMohb2qpb4rbisFC4dtnWyQeheu6EgrbnpGHBc1mkzYbxkqb",
8015],
  ["6PAzwkAhjNPFrXDBE2Ew9YovvAZ3Z3abFEA6xFuRAgRDT2dsNyxDvrLLX7HVQwWht7",
7968],
  ["7FM7CE9wLpbFn317a7LcPvTRCcBx4jPb4NMmnM9zngo42JNK9aA1SETBufE6VkWaFj",
7952],
  ["5wXoFWTZAfrGZEyL2TzWGr2XQGSgdqbEbTGTWdhX9ciNEyh8yJoNWV7zBbWP2YcYYY",
7932],
  ["6YCDJ4f6dD8c5WuxDRp82ZYdho2Ha5qSeFTAZ3688y485hzH1WKViWwtFXm2qKdQEu",
7912],
```

```
  ["7SGTxaLStDPbLKnjcXC1fqN3xaGBs4oFEPVa6UVDwt5N63C7US4vf8wP9hB4p9Fdku",
7674],
  ["6yGYBgPmnvRfL8PuEECxkzpjCkdYwyTUP5abeqzKBB7BYfb2nU3YkUzC1KBNLwJRFd",
7605],
  ["78LByD8m22fAkAUxmpWf67WihHCeHX3SW2URN5H7zN2gqBdDy86LrKVxU9NR7DgCdK",
7590],
  ["5nooK1aeaUdD84oFsjUymzCiHRWNHzD6NBY5CSrZgSFNRwQUj7h3jA9gf62FcPPRWb",
7484],
  ["7HmEZ2zBKAaky3pfa3BjNiqDMoFgdzBpdkZfM5fH96hh8DhWE8nwpSPFE5KHMMaAmG",
7348],
  ["6PBwYc68quDCZFvi3qDwVHwEWWHUasAYZtbwzm6RX189pLXR7m8G3gDCNRMmVxxYJY",
7339],
  ["5t1jmA1tc68UFrBBfyGFJ6DUEpXUfUAMUfd65YqzfMkX7TcG2MvcNvoWMEyVh4tuYq",
7151],
  ["5kfTrgNrVFzkGAkQXDsPTreinAyPEE9szp8jvkBv44UnyHjpEZLm1xWpNNvLMSHCg9",
6998],
  ["6bmQpVDCyPBj3eNQ4Bz7zjGCC52rZ2PgtBANazbkESPVjxoHyGkJuPmmyB15ikekTH",
4904],
  ["72F3MbVLuUzxALsXAsTEJptYFGL9ewrHFhktZCGvj7JhWXnKJScpVqbbVTAEvU9Tok",
167],
  ["77PbahMBp91VnyQsBc3f59JvEiGoSuS6JUUtYmm1i3XZHxYyekgNoaoifJ5NLHA7sJ",
139]
]
```

You make it as validator into the next epoch. Your goal now is to produce entries on-time within the 500ms window until the epoch ends.

If you fail to produce an entry on-time or are stalling the network by producing late entries you are candidate for slashing.

If you:

- Over the last 10 entries your average slot time is higher than 700ms
- Do not produce an entry when it is your slot within 8seconds

Any validator can now call an on-chain special meeting and produce the proof plus agenda with the topic to vote for your removal, if consensus from other validators is reached via voting you are immediately dropped from the epoch and forfeit all accumulated score, any further calculations at epoch-end count your score as if it was 0.

We are thinking to make this less harsh as there has been cases where honest nodes were being attacked and illegally DOSed to bring them offline and make candidates for slashing out of them. The benefit was their score was removed from the calculation of the emission so the attacker got a bigger piece for themselves at epoch end.

EXCHANGES

Trustless Flow

You can run an AMA integration using purely the RPC and its quite reliable.

You also need to download the latest binary release of the node and run it as `OFFLINE=true ./amadeusd`, it functions as a utility tool similar to `geth`.

Get the current chain height and hash from your integrations inception:

```
curl -s https://nodes.amadeus.bot/api/chain/stats \
| jq -r '.stats.tip.header_unpacked.height'

34959084

curl -s https://nodes.amadeus.bot/api/chain/stats \
| jq -r '.stats.tip.hash'

BENtrybSL5UGBnjJMm7sVZ9fr1CN7Qu9R3SdH88GxdEt
```

Next generate deposit wallets as required:

```
./amadeusd --bw-command start generate_wallet

5ibhX6qynQvFuBaQH1phfqCCY3VKMw4kLXkcSeBCNeAyCpyMVYzsEWx9FNBu6yz2XF
3CHRgeHNDzg3qjnFGdruvXqVb45TAWXzGZpnK2mKKHaByy7ZiQUMaDjLbQu1z4AnfsNtN2diM
6Tc9PBVPgWZ7zgL
```

Check for finality and scan the block for deposits or withdrawls:

```
curl -s https://nodes.amadeus.bot/api/chain/hash/
BENtrybSL5UGBnjJMm7sVZ9fr1CN7Qu9R3SdH88GxdEt?filter_on_function=transfer
\
| jq -r '.entry.consensus.finality_reached'
true

curl -s https://nodes.amadeus.bot/api/chain/hash/
BENtrybSL5UGBnjJMm7sVZ9fr1CN7Qu9R3SdH88GxdEt?filter_on_function=transfer
\
| jq -r '.entry.txs_filtered'
[
  {
    "metadata": {
      "entry_hash": "BENtrybSL5UGBnjJMm7sVZ9fr1CN7Qu9R3SdH88GxdEt",
      "entry_slot": 34959084
    },
    "signature":
"231XbxmVFKYNJsFb8X49wGAx1HVbW7FuiNFzVxuF1BCey5SpdsRzaSYRzTJEdJ22XF8g3Nfz
JgqnMrgypxZYbVV3hJWVTqRukY6C6cnPYNDbUTyu2gHVyQvNqRUkNfCmmi58",
    "hash": "6NxepwR3YTf3JbNoTtycZ73eLmY79CFzhueFJdYvDsoE",
    "tx": {
      "nonce": 1760628863344857048,
      "signer":
"69TDon8KJp3vicNeFR3dg5x5sKY8PJFLmoizX3RN31YL4fwr266AVcXgwy1mjCLy6M",
      "actions": [
        {
          "args": [

"5tUw2maGeGEvNyn6hkwssyNgiEd7K4Dp6jtkDFTSkbHZGiWR4Q12JMpbtC3n3r8cef",
            "645160000000000",
            "AMA"
          ],
          "function": "transfer",
          "op": "call",
          "contract": "Coin"
        }
      ]
    }
  }
]
```

Parse the single relevant transfer transaction now:

```
 Sender:
"69TDon8KJp3vicNeFR3dg5x5sKY8PJFLmoizX3RN31YL4fwr266AVcXgwy1mjCLy6M",
 Receiver:
"5tUw2maGeGEvNyn6hkwssyNgiEd7K4Dp6jtkDFTSkbHZGiWR4Q12JMpbtC3n3r8cef"
 Amount: 64516000000000 (9 decimals)
 AmountAsFloat: 64516.0  (64516000000000 / 1_000_000_000)
```

Check if there is another block after that has reached finality and repeat.

```
curl -s https://nodes.amadeus.bot/api/chain/hash/
BENtrybSL5UGBnjJMm7sVZ9fr1CN7Qu9R3SdH88GxdEt?filter_on_function=transfer
\
| jq -r '.entry.next_entry_hash_finality_reached'
789CfwfidnT44vVgGWz7xnm8CB7VLxBYMkUPeGAysWD2
```

To process a withdrawl now create a serialized transaction substituting the SEED64= envvariable for your `sender seed/privatekey` as base58, then 69receiver as the receivers public key in base58, amount here cannot be float it must be the full integer accounting for 9 decimals.

```
SEED64=sender ./amadeusd --bw-command start buildtx Coin transfer
["69receiver", "1000000000", "AMA"]

jKhTr4KjP2rsAGvtHfsX7h5MbBeKt1WbL78N3KVofpRc4f85UBjNhdMJH9AndZLEQpRsg5BcZ
LQ
CGiYCK7k7hchJcmmBh4cjZDkWzSSpcHenqZrb5tKPRdXDx1Xin13MzKQPukmGDQzgMUzkC2Rz
Qv
DrJJmY7CJqcXPF7QCJrAuHBqyv79ysGnRugkReHwkCRyd99hXwkWeFYHeS1aBLAntpmFwqzc2
bD
ANTms9Har4WgEX3pDs32mwCDsTJ3nYPbRAee1Gp77CuZPa9fQsVEGoskFz5xrf2kCXjBG8Nhd
N1
X5NNfGV5vZDYDt9J3oYG1Z3Z6yWzbv5pbhnkMupiFShSAiXssVa2H7MhozQ2xrtxyNhrtHQGr
d5
NAA8dnfNCXYQ2enHtJFFzg6vP2Zajye87BwCxvVv4TELjBKs1ajBqVrrMiT411KhAL1yn1NnT
ji
RjhEfRyPvh5Mpnu1YoAVvhQFtYCUMWK9y7

%{
  signature: <<165, 29, 132, 117, 78, ...>>,
  hash: <<224, 102, 230, 139, 81, 81, ...>>,
  tx: %{
    nonce: 1761339817312206803,
    signer: <<180, 138, 83, 27, 27, ...>,
    actions: [%{
      args: ["69receiver", "1000000000", "AMA"],
      function: "transfer",
      op: "call",
      contract: "Coin"}]
  },
  tx_encoded: <<7, 1, 3, 5, ...>>
}
```

Call a similar API `build_and_broadcasttx` to directly broadcast it to an RPC node:

```
SEED64=sender ./amadeusd --bw-command start build_and_broadcasttx Coin
transfer ["69receiver", "1000000000", "AMA"]

78VMtHDJCtmFohiRNAS89g3pQNvhmtN6KZ4jyGhe74Xr
```

You can now check the TX hash to confirm the RPC node relayed it to a validator and it got included in the chain

```
curl -s https://nodes.amadeus.bot/api/chain/
tx/78VMtHDJCtmFohiRNAS89g3pQNvhmtN6KZ4jyGhe74Xr | jq
{
  "metadata": {
    "entry_hash": "8k3HVDQ6cqrbFr8hPcdZV1Tos6UykAuMhs6oaJeV1MbP",
    "entry_slot": 36729887
  },
  "signature":
"25iZprHk5MZiZi6CXKLFqM5QNJn2UGxLwKD5ZpiLWJH7Y4h1ai3uJgXsjhczipGW2W2fzCEz
ft8pneiuG4cRojXnfJ1XeWyAEXMAPQh3Xb1ojJaUV4XyF8iWa6gZgDosdy3i",
  "result": {
    "error": "ok"
  },
  "hash": "78VMtHDJCtmFohiRNAS89g3pQNvhmtN6KZ4jyGhe74Xr",
  "tx": {
    "nonce": 28612872,
    "signer":
"6KFVHM35azmepNw4MEzDPb65WhjU28oJjTaz9V7QYeCCTVAASHzFUhUiMC2njmUwCS",
    "actions": [
      {
        "args": [
          "wqvhdreKnJ3fN45PqmoBwMC..."
        ],
        "function": "submit_sol",
        "op": "call",
        "contract": "Epoch"
      }
    ]
  }
}
```

The only part missing is to prove the RPC is not lying in a trustless way. This is beyond the scope right now of this guide and protocol upgrades are underway as well.

Of course the best way now is to run a personal node and you can follow the validator guide for that.

Advanced Verification

Currently this page is under construction. It will contain how to verify data without running a full node by checking proofs.

HTTP API

WebSocket

WebSocket RPC endpoints for real-time communication

WebSocket RPC endpoint

GET https://nodes.amadeus.bot/ws/rpc

Upgrade HTTP connection to WebSocket for real-time RPC communication

Header parameters

Upgrade `string · enum` required

Possible values: `websocket`

Connection `string · enum` required

Possible values: `Upgrade`

Responses

101 Switching Protocols - WebSocket connection established

No content

400 Bad Request - Invalid WebSocket handshake

GET /ws/rpc HTTP

```
GET /ws/rpc HTTP/1.1
Host: nodes.amadeus.bot
Upgrade: websocket
Connection: Upgrade
Accept: */*
```

Test it

101 Switching Protocols - WebSocket connection established

No content

WebSocket test page

GET https://nodes.amadeus.bot /ws/rpc/test

Returns an HTML test page for WebSocket RPC functionality

Responses

200 HTML test page text/html

GET /ws/rpc/test HTTP

```
GET /ws/rpc/test HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 HTML test page

```
text
```

Peer

Network peer and validator information

Get ANR by public key

`GET` `https://nodes.amadeus.bot` `/api/peer/anr/{pk}`

Retrieve Address and Routing (ANR) information for a specific peer by public key

Path parameters

pk `string` required

Public key (Base58 encoded)

Responses

> `200` ANR information `application/json`

`GET` `/api/peer/anr/{pk}` HTTP

```
GET /api/peer/anr/{pk} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it ▶

`200` ANR information

```
{
  "error": "ok",
  "anr": {}
}
```

Get ANRs for all validators

GET https://nodes.amadeus.bot /api/peer/anr_validators

Retrieve Address and Routing information for all active validators

Responses

200 List of validator ANRs application/json

GET /api/peer/anr_validators HTTP

```
GET /api/peer/anr_validators HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 List of validator ANRs

```
{
  "error": "ok",
  "anrs": [
    {}
  ]
}
```

Get all ANRs

`GET` `https://nodes.amadeus.bot` `/api/peer/anr`

Retrieve Address and Routing information for all known peers

Responses

> `200` List of all ANRs `application/json`

`GET` `/api/peer/anr` HTTP

```
GET /api/peer/anr HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` List of all ANRs

```
{
  "error": "ok",
  "anrs": [
    {}
  ]
}
```

Get all nodes

GET https://nodes.amadeus.bot /api/peer/nodes

Retrieve information about all connected nodes

Responses

200 List of nodes application/json

GET /api/peer/nodes HTTP

```
GET /api/peer/nodes HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 List of nodes


```
{
  "error": "ok",
  "nodes": [
    {}
  ]
}
```

Get all trainers (validators)

GET https://nodes.amadeus.bot /api/peer/trainers

Retrieve list of all active validator nodes

Responses

200 List of trainers application/json

GET /api/peer/trainers HTTP

```
GET /api/peer/trainers HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 List of trainers

```
{
  "error": "ok",
  "trainers": [
    {}
  ]
}
```

Get removed trainers

GET https://nodes.amadeus.bot /api/peer/removed_trainers

Retrieve list of validators that have been removed

Responses

200 List of removed trainers application/json

GET /api/peer/removed_trainers HTTP

```
GET /api/peer/removed_trainers HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 List of removed trainers

```
{
  "error": "ok",
  "removed_trainers": [
    {}
  ]
}
```

Chain

Blockchain data and statistics

Get chain statistics

GET https://nodes.amadeus.bot/api/chain/stats

Retrieve current blockchain statistics including height, validators, etc.

Responses

200 Chain statistics application/json

GET /api/chain/stats HTTP

```
GET /api/chain/stats HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 Chain statistics


```
{
  "error": "ok",
  "stats": {
    "height": 1,
    "validators": 1,
    "epoch": 1
  }
}
```

Get latest entry (tip)

GET https://nodes.amadeus.bot /api/chain/tip

Retrieve the most recent blockchain entry

Responses

200 Latest entry application/json

GET /api/chain/tip HTTP

```
GET /api/chain/tip HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 Latest entry

```
{}
```

Get entry by hash

`GET` `https://nodes.amadeus.bot` `/api/chain/hash/{hash}`

Retrieve a blockchain entry by its hash

Path parameters

hash `string` required

Entry hash (Base58 encoded)

Query parameters

filter_on_function `string` optional

Filter transactions by contract function name

Responses

200 Entry data application/json

`GET` `/api/chain/hash/{hash}` HTTP

```
GET /api/chain/hash/{hash} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 Entry data

```
{}
```

Get entry by height

`GET` `https://nodes.amadeus.bot` `/api/chain/height/{height}`

Retrieve a blockchain entry by its height

Path parameters

height `integer` required

Block height

Responses

> `200` Entry data `application/json`

`GET` `/api/chain/height/{height}` HTTP

```
GET /api/chain/height/{height} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it ▶

`200` Entry data

```
{}
```

Get entry by height with full transactions

`GET` `https://nodes.amadeus.bot` `/api/chain/height_with_txs/{height}`

Retrieve a blockchain entry by height including full transaction details

Path parameters

height `integer` required

Block height

Responses

`200` Entry data with transactions `application/json`

`GET` `/api/chain/height_with_txs/{height}` HTTP

```
GET /api/chain/height_with_txs/{height} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` Entry data with transactions

```
{}
```

Get transaction by ID

GET https://nodes.amadeus.bot /api/chain/tx/{txid}

Retrieve a transaction by its ID

Path parameters

txid string required

Transaction ID (Base58 encoded)

Responses

200 Transaction data application/json

GET /api/chain/tx/{txid} HTTP

```
GET /api/chain/tx/{txid} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 Transaction data

```
{}
```

Get transaction events by account

GET `https://nodes.amadeus.bot` `/api/chain/tx_events_by_account/{account}`

Retrieve transaction history for a specific account with pagination and filtering

Path parameters

account `string` required

Account address (Base58 encoded)

Query parameters

limit `integer` optional

Maximum number of transactions to return

Default: `100`

offset `integer` optional

Number of transactions to skip

Default: `0`

sort `string · enum` optional

Sort order

Default: `asc`

Possible values: `asc` `desc`

type `string · enum` optional

Filter by transaction type

Possible values: `sent` `recv`

cursor `string` optional

Pagination cursor (raw bytes)

cursor_b58 `string` optional

Pagination cursor (Base58 encoded)

contract `string` optional

Filter by contract address (raw bytes)

contract_b58 `string` optional

Filter by contract address (Base58 encoded)

function `string` optional

Filter by contract function name

GET /api/chain/tx_events_by_account/{account} HTTP

```
GET /api/chain/tx_events_by_account/{account} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it ▶

200 Transaction events


```
{
  "cursor": "text",
  "txs": [
    {}
  ]
}
```

Get transactions in entry

`GET` `https://nodes.amadeus.bot` `/api/chain/txs_in_entry/{entry_hash}`

Retrieve all transactions contained in a specific entry

Path parameters

entry_hash `string` required

Entry hash (Base58 encoded)

Responses

`200` List of transactions `application/json`

`GET` `/api/chain/txs_in_entry/{entry_hash}` HTTP

```
GET /api/chain/txs_in_entry/{entry_hash} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` List of transactions

```
{
  "error": "ok",
  "txs": [
    {}
  ]
}
```

Epoch

Epoch and validator scoring information

Get epoch scores for all validators

GET https://nodes.amadeus.bot/api/epoch/score

Retrieve mining scores for all validators in the current epoch

Responses

200 Epoch scores application/json

GET /api/epoch/score HTTP

```
GET /api/epoch/score HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 Epoch scores

```
{}
```

Get epoch score for validator

`GET` `https://nodes.amadeus.bot` `/api/epoch/score/{pk}`

Retrieve mining score for a specific validator

Path parameters

pk `string` required

Validator public key (Base58 encoded)

Responses

`200` Validator epoch score `application/json`

`GET` `/api/epoch/score/{pk}` HTTP

```
GET /api/epoch/score/{pk} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` Validator epoch score

```
{}
```

Get emission address for validator

`GET` `https://nodes.amadeus.bot` `/api/epoch/get_emission_address/{pk}`

Retrieve the emission (reward) address for a validator

Path parameters

pk `string` required

Validator public key (Base58 encoded)

Responses

> `200` Emission address `application/json`

`GET` `/api/epoch/get_emission_address/{pk}` HTTP

```
GET /api/epoch/get_emission_address/{pk} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it ▶

`200` Emission address

```
{
  "error": "ok",
  "emission_address": "text"
}
```

Check if solution is in epoch

`GET` `https://nodes.amadeus.bot` `/api/epoch/sol_in_epoch/{epoch}/{sol_hash}`

Verify if a specific proof-of-work solution was submitted in an epoch

Path parameters

epoch `integer` required

Epoch number

sol_hash `string` required

Solution hash (Base58 encoded)

Responses

`200` Solution verification result `application/json`

`GET` `/api/epoch/sol_in_epoch/{epoch}/{sol_hash}` HTTP

```
GET /api/epoch/sol_in_epoch/{epoch}/{sol_hash} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` Solution verification result

```
{}
```

Contract

Smart contract operations and state queries

Validate WASM contract

POST https://nodes.amadeus.bot /api/contract/validate

Validate a WASM contract bytecode before deployment

Body

application/octet-stream

string · binary optional

WASM bytecode

Responses

200 Validation result application/json

POST /api/contract/validate HTTP

```
POST /api/contract/validate HTTP/1.1
Host: nodes.amadeus.bot
Content-Type: application/octet-stream
Accept: */*
Content-Length: 8

"binary"
```

Test it

200 Validation result

```
{
  "error": "text",
  "valid": true
}
```

Get contract state by key

POST https://nodes.amadeus.bot /api/contract/get

Retrieve a value from contract state storage

Body

application/octet-stream

string · binary optional

Storage key (raw bytes)

Responses

200 Contract state value application/json

POST /api/contract/get HTTP

```
POST /api/contract/get HTTP/1.1
Host: nodes.amadeus.bot
Content-Type: application/octet-stream
Accept: */*
Content-Length: 8

"binary"
```

Test it

200 Contract state value

```
{}
```

Get contract state by prefix

POST https://nodes.amadeus.bot /api/contract/get_prefix

Retrieve all key-value pairs from contract state with a given prefix

Body

application/octet-stream

string · binary optional

Storage key prefix (raw bytes)

Responses

200 Contract state entries (vecpak encoded) application/octet-stream

POST /api/contract/get_prefix HTTP

```
POST /api/contract/get_prefix HTTP/1.1
Host: nodes.amadeus.bot
Content-Type: application/octet-stream
Accept: */*
Content-Length: 8

"binary"
```

Test it

200 Contract state entries (vecpak encoded)

```
binary
```

Execute contract view function (GET)

`GET` `https://nodes.amadeus.bot` `/api/contract/view`

Execute a read-only contract function with no arguments

Path parameters

`contract` `string` required

Contract address (Base58 encoded)

`function` `string` required

Function name

Query parameters

`pk` `string` optional

Caller public key (Base58 encoded)

Responses

`200` Function execution result `application/json`

GET /api/contract/view HTTP

```
GET /api/contract/view HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it ▶

200 Function execution result


```
{
  "success": true,
  "result": "text",
  "logs": [
    "text"
  ]
}
```

Execute contract view function (vecpak)

POST https://nodes.amadeus.bot /api/contract/view

Execute a read-only contract function using vecpak encoding

Body

application/octet-stream

string · binary optional

Vecpak encoded map with fields:

- contract: Contract address (48 bytes)
- function: Function name (string)
- args: Function arguments (array)
- pk: Optional caller public key (48 bytes)

Responses

200 Function execution result application/json

POST /api/contract/view HTTP

```
POST /api/contract/view HTTP/1.1
Host: nodes.amadeus.bot
Content-Type: application/octet-stream
Accept: */*
Content-Length: 8

"binary"
```

Test it ▶

200 Function execution result

```
{
  "success": true,
  "result": "text",
  "logs": [
    "text"
  ]
}
```

Get richlist

`GET` `https://nodes.amadeus.bot` `/api/contract/richlist`

Retrieve the list of top AMA token holders

Responses

> `200` Richlist `application/json`

`GET` `/api/contract/richlist` HTTP

```
GET /api/contract/richlist HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it ▶

`200` Richlist


```
{
  "error": "ok",
  "richlist": [
    {
      "address": "text",
      "balance": "text"
    }
  ]
}
```

Wallet

Wallet balance queries

Get AMA balance

`GET` `https://nodes.amadeus.bot` `/api/wallet/balance/{pk}`

Retrieve AMA token balance for an account

Path parameters

pk `string` required

Public key (Base58 encoded)

Responses

> `200` Balance `application/json`

`GET` `/api/wallet/balance/{pk}` HTTP

```
GET /api/wallet/balance/{pk} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` Balance

```
{
  "error": "ok",
  "balance": "text"
}
```

Get token balance

`GET` `https://nodes.amadeus.bot` `/api/wallet/balance/{pk}/{symbol}`

Retrieve token balance for a specific asset

Path parameters

pk `string` required

Public key (Base58 encoded)

symbol `string` required

Token symbol (e.g., AMA)

Responses

200 Balance application/json

`GET` `/api/wallet/balance/{pk}/{symbol}` HTTP

```
GET /api/wallet/balance/{pk}/{symbol} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 Balance

```
{
  "error": "ok",
  "balance": "text"
}
```

Get all balances

GET https://nodes.amadeus.bot /api/wallet/balance_all/{pk}

Retrieve all token balances for an account

Path parameters

pk `string` required

Public key (Base58 encoded)

Responses

200 All balances application/json

GET /api/wallet/balance_all/{pk} HTTP

```
GET /api/wallet/balance_all/{pk} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

200 All balances

```
{
  "error": "ok",
  "balances": {
    "ANY_ADDITIONAL_PROPERTY": "text"
  }
}
```

Transaction

Transaction submission and retrieval

Submit transaction (POST)

POST https://nodes.amadeus.bot /api/tx/submit

Submit a signed transaction to the network

Body

application/octet-stream

string · binary optional

Packed transaction (raw bytes or Base58 encoded)

Responses

200 Submission result application/json

POST /api/tx/submit HTTP

```
POST /api/tx/submit HTTP/1.1
Host: nodes.amadeus.bot
Content-Type: application/octet-stream
Accept: */*
Content-Length: 8

"binary"
```

Test it

200 Submission result

```
{
  "error": "text",
  "txid": "text"
}
```

Submit transaction (GET)

`GET` `https://nodes.amadeus.bot` `/api/tx/submit/{tx_packed}`

Submit a signed transaction using GET method

Path parameters

tx_packed `string` required

Packed transaction (Base58 encoded)

Responses

`200` Submission result `application/json`

`GET` `/api/tx/submit/{tx_packed}` HTTP

```
GET /api/tx/submit/{tx_packed} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` Submission result

```
{}
```

Submit and wait for confirmation (POST)

POST https://nodes.amadeus.bot /api/tx/submit_and_wait

Submit a transaction and wait for it to be included in a block

Body

application/octet-stream

string · binary optional

Packed transaction (raw bytes or Base58 encoded)

Responses

200 Submission and confirmation result application/json

POST /api/tx/submit_and_wait HTTP

```
POST /api/tx/submit_and_wait HTTP/1.1
Host: nodes.amadeus.bot
Content-Type: application/octet-stream
Accept: */*
Content-Length: 8

"binary"
```

Test it

200 Submission and confirmation result

```
{}
```

Submit and wait for confirmation (GET)

`GET` `https://nodes.amadeus.bot` `/api/tx/submit_and_wait/{tx_packed}`

Submit a transaction and wait for confirmation using GET method

Path parameters

tx_packed `string` required

Packed transaction (Base58 encoded)

Responses

> `200` Submission and confirmation result `application/json`

`GET` `/api/tx/submit_and_wait/{tx_packed}` HTTP

```
GET /api/tx/submit_and_wait/{tx_packed} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it ▶

`200` Submission and confirmation result

```
{}
```

Proof

Cryptographic proofs for verification

Get validator proof for entry

`GET` `https://nodes.amadeus.bot` `/api/proof/validators/{entry_hash}`

Get cryptographic proof of validator attestations for an entry

Path parameters

`entry_hash` `string` required

Entry hash (Base58 encoded)

Responses

> `200` Validator proof `application/json`

`GET` `/api/proof/validators/{entry_hash}` HTTP

```
GET /api/proof/validators/{entry_hash} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` Validator proof

```
{}
```

Get contract state proof

`GET` `https://nodes.amadeus.bot` `/api/proof/contractstate/{key}`

Get cryptographic proof for a contract state key

Path parameters

key `string` required

Contract state key (Base58 encoded)

Responses

`200` State proof `application/json`

`GET` `/api/proof/contractstate/{key}` HTTP

```
GET /api/proof/contractstate/{key} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` State proof

```
{}
```

Get contract state proof with value

`GET` `https://nodes.amadeus.bot` `/api/proof/contractstate/{key}/{value}`

Get cryptographic proof for a contract state key-value pair

Path parameters

key `string` required

Contract state key (Base58 encoded)

value `string` required

Expected value (Base58 encoded)

Responses

`200` State proof `application/json`

`GET` `/api/proof/contractstate/{key}/{value}` HTTP

```
GET /api/proof/contractstate/{key}/{value} HTTP/1.1
Host: nodes.amadeus.bot
Accept: */*
```

Test it

`200` State proof

```
{}
```

Get contract state proof (POST)

POST https://nodes.amadeus.bot /api/proof/contractstate

Get cryptographic proof for contract state using vecpak encoding

Body

application/octet-stream

string · binary optional

Vecpak encoded map with fields:

- key: State key (bytes)
- value: Optional expected value (bytes)

Responses

200 State proof application/json

POST /api/proof/contractstate HTTP

```
POST /api/proof/contractstate HTTP/1.1
Host: nodes.amadeus.bot
Content-Type: application/octet-stream
Accept: */*
Content-Length: 8

"binary"
```

Test it

200 State proof

```
{}
```

Models

Error +

Contribution Guide

Status: Internal contributions only at this stage. Public contribution flow to be opened once Nova SDK is released. Below is a draft structure for when contributions go live.

Branching Strategy

- Main development occurs on dev.
- Releases are merged into main and tagged per semantic versioning.
- Feature branches follow the format: feature/<short-description>

Commit Format

- Uses Conventional Commits.
- Example:

feat(runtime): add MatMul validation for uPoW agent

fix(graph): correct version conflict in Living System Graph

Pull Request Checklist

- Code builds and passes all tests
- Linting is clean (see tools below)
- Relevant tests added or updated
- Associated documentation updated
- Reviewed by at least one core team member

Code Style & Tooling

- rustfmt.toml for Rust formatting
- .editorconfig included at root level
- Future support for Python linting: black, ruff, mypy (TBD)
- Pre-commit hooks recommended (to be provided in SDK)

Contributor License Agreement (CLA)

- Not yet required — public contributions are not enabled.
- CLA process will be outlined ahead of SDK and Graph API release.