NeoRust SDK Documentation

Neo Logo   Neo X Logo   R3E Logo

Welcome to the NeoRust SDK documentation. This SDK provides a comprehensive set of tools for interacting with the Neo N3 blockchain using Rust.

Features

- **Wallet Management**: Create, load, and manage Neo wallets
- **Transaction Building**: Build and sign transactions
- **Smart Contract Interaction**: Deploy and invoke smart contracts
- **NEP-17 Token Support**: Interact with NEP-17 tokens
- **NNS Integration**: Work with the Neo Name Service
- **Famous Contract Support**: Direct interfaces for Flamingo, NeoburgerNeo, GrandShare, and NeoCompound
- **Neo X Support**: EVM compatibility and bridge functionality
- **SGX Support**: Secure operations within Intel SGX enclaves

Getting Started

To get started with the NeoRust SDK, see the Getting Started Guide.

API Reference

For detailed API documentation, see the API Reference.

Examples

Check out the Examples section for code examples demonstrating various features of the SDK.

Getting Started with NeoRust SDK

This guide will help you get started with the NeoRust SDK for interacting with the Neo N3 blockchain.

Prerequisites

- Rust 1.70.0 or later
- Cargo package manager
- Basic knowledge of the Neo blockchain

Installation

Add the NeoRust SDK to your Cargo.toml:

```toml
[dependencies]
neo = { git = "https://github.com/R3E-Network/NeoRust" }
```

Or if you prefer to use a specific version:

```toml
[dependencies]
neo = "0.1.0"
```

Basic Usage

Here's a simple example of connecting to a Neo N3 node and getting the current block height:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Get the current block height
    let block_count = provider.get_block_count().await?;
    println!("Current block height: {}", block_count);

    Ok(())
}
```

Next Steps

- Learn about Wallet Management
- Explore Smart Contract Interaction
- See Examples for more code samples

Installation Guide

This guide provides detailed instructions for installing and configuring the NeoRust SDK.

Requirements

- Rust 1.70.0 or later
- Cargo package manager
- Optional: Intel SGX SDK (for SGX features)

Standard Installation

Add the NeoRust SDK to your Cargo.toml:

```toml
[dependencies]
neo = { git = "https://github.com/R3E-Network/NeoRust" }
```

Feature Flags

NeoRust supports various feature flags to enable specific functionality:

```toml
[dependencies]
neo = { git = "https://github.com/R3E-Network/NeoRust", features = ["ledger", "aws"] }
```

Available features:

- `ledger`: Support for Ledger hardware wallets

- `aws` : AWS integration
- `sgx` : Intel SGX support (requires additional setup)

SGX Support

To use the SGX features, you need to install the Intel SGX SDK. See the SGX Setup Guide for detailed instructions.

Verifying Installation

To verify that the SDK is installed correctly, create a simple test program:

```rust
use neo::prelude::*;

fn main() {
    println!("NeoRust SDK installed successfully!");
}
```

Compile and run the program:

```
cargo run
```

If the program compiles and runs without errors, the SDK is installed correctly.

Wallet Management

This tutorial covers wallet management with the NeoRust SDK, including creating, loading, and using wallets.

Creating a New Wallet

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Create a new wallet with a password
    let password = "my-secure-password";
    let wallet = Wallet::new(password)?;

    // Generate a new account
    let account = wallet.create_account()?;
    println!("New account address: {}", account.address());

    // Save the wallet to a file
    wallet.save("my-wallet.json")?;

    Ok(())
}
```

Loading an Existing Wallet

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Load a wallet from a file
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the default account
    let account = wallet.default_account()?;
    println!("Default account address: {}",
account.address());

    Ok(())
}
```

Importing a Private Key

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Create a new wallet
    let password = "my-secure-password";
    let mut wallet = Wallet::new(password)?;

    // Import a private key
    let private_key = "your-private-key-here";
    let account = wallet.import_private_key(private_key,
password)?;
    println!("Imported account address: {}",
account.address());

    // Save the wallet
    wallet.save("my-wallet.json")?;

    Ok(())
}
```

Signing a Message

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Load a wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the default account
    let account = wallet.default_account()?;

    // Sign a message
    let message = b"Hello, Neo!";
    let signature = account.sign_message(message)?;
    println!("Signature: {:?}", signature);

    // Verify the signature
    let is_valid = account.verify_signature(message,
&signature)?;
    println!("Signature valid: {}", is_valid);

    Ok(())
}
```

Wallet Backup and Recovery

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Create a new wallet
    let password = "my-secure-password";
    let wallet = Wallet::new(password)?;

    // Generate a new account
    let account = wallet.create_account()?;

    // Get the mnemonic phrase for backup
    let mnemonic = wallet.export_mnemonic(password)?;
    println!("Backup phrase: {}", mnemonic);

    // Later, recover the wallet from the mnemonic
    let recovered_wallet = Wallet::from_mnemonic(&mnemonic,
password)?;

    // Verify the recovered wallet has the same account
    let recovered_account =
recovered_wallet.default_account()?;
    println!("Recovered account address: {}",
recovered_account.address());

    assert_eq!(account.address(),
recovered_account.address());

    Ok(())
}
```

Using SGX-Protected Wallets

If you have enabled the SGX feature, you can use SGX-protected wallets for enhanced security:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Initialize the SGX enclave
    let enclave_path = "path/to/enclave.so";
    let enclave_manager =
SgxEnclaveManager::new(enclave_path)?;

    // Create a wallet with a password
    let password = "my-secure-password";
    let wallet = enclave_manager.create_wallet(password)?;

    // Get the wallet's public key
    let public_key = wallet.get_public_key();
    println!("Wallet public key: {:?}", public_key);

    // Sign a transaction using the SGX-protected private key
    let transaction_data = b"Sample transaction data";
    let signature =
wallet.sign_transaction(transaction_data)?;

    Ok(())
}
```

Best Practices

1. **Secure Password Storage**: Never hardcode passwords in your application.
2. **Regular Backups**: Always backup your wallet's mnemonic phrase or private keys.
3. **Verify Addresses**: Always verify addresses before sending transactions.
4. **Use Hardware Wallets**: For production applications, consider using hardware wallets via the Ledger feature.
5. **SGX Protection**: For high-security applications, use SGX-protected wallets.

Smart Contracts

This tutorial covers working with smart contracts on the Neo N3 blockchain using the NeoRust SDK.

Understanding Neo Smart Contracts

Neo N3 smart contracts are written in a variety of languages including C#, Python, Go, and TypeScript, and are compiled to NeoVM bytecode. The NeoRust SDK provides tools for deploying and interacting with these contracts.

Deploying a Smart Contract

To deploy a smart contract, you need the contract's NEF (Neo Executable Format) file and its manifest:

```rust
use neo::prelude::*;
use std::path::Path;
use std::fs;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will deploy the contract
    let account = wallet.default_account()?;

    // Read the NEF file and manifest
    let nef_bytes = fs::read("path/to/contract.nef")?;
    let manifest_bytes =
fs::read("path/to/contract.manifest.json")?;

    // Create a transaction to deploy the contract
    let transaction = TransactionBuilder::new()
        .version(0)
        .nonce(rand::random::<u32>())
        .valid_until_block(provider.get_block_count().await?
+ 100)
        .deploy_contract(&nef_bytes, &manifest_bytes)
        .sign(account)?
        .build();

    // Send the transaction
    let txid =
provider.send_raw_transaction(&transaction).await?;
    println!("Contract deployed with transaction ID: {}",
txid);

    // Wait for the transaction to be confirmed
    let receipt = provider.wait_for_transaction(&txid, 60,
2).await?;
    println!("Transaction confirmed: {:?}", receipt);
```

```
        Ok(())
    }
```

Invoking a Smart Contract

Once a contract is deployed, you can invoke its methods:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will invoke the contract
    let account = wallet.default_account()?;

    // Contract script hash (address)
    let contract_hash =
"0x1234567890abcdef1234567890abcdef12345678".parse::
<ScriptHash>()?;

    // Create a transaction to invoke the contract
    let transaction = TransactionBuilder::new()
        .version(0)
        .nonce(rand::random::<u32>())
        .valid_until_block(provider.get_block_count().await?
+ 100)
        .script(
            ScriptBuilder::new()
                .contract_call(
                    contract_hash,
                    "transfer",
                    &[

ContractParameter::hash160(account.address().script_hash()),

ContractParameter::hash160("NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAa
j".parse::<Address>()?),
                        ContractParameter::integer(1000),
                        ContractParameter::any(None),
                    ],
                )
                .to_array()
        )
```

```rust
        .sign(account)?
        .build();

    // Send the transaction
    let txid =
provider.send_raw_transaction(&transaction).await?;
    println!("Contract invoked with transaction ID: {}",
txid);

    // Wait for the transaction to be confirmed
    let receipt = provider.wait_for_transaction(&txid, 60,
2).await?;
    println!("Transaction confirmed: {:?}", receipt);

    Ok(())
}
```

Reading Contract State

You can read the state of a contract without sending a transaction:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Contract script hash (address)
    let contract_hash =
"0x1234567890abcdef1234567890abcdef12345678".parse::
<ScriptHash>()?;

    // Address to check balance for
    let address =
"NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAaj".parse::<Address>()?;

    // Invoke the contract's balanceOf method
    let result = provider.invoke_function(
        contract_hash,
        "balanceOf",
        &[ContractParameter::hash160(address.script_hash())],
        None,
    ).await?;

    // Parse the result
    if let Some(stack) = result.stack.first() {
        if let Some(value) = stack.as_integer() {
            println!("Balance: {}", value);
        }
    }

    Ok(())
}
```

Working with NEP-17 Tokens

NEP-17 is Neo's token standard, similar to Ethereum's ERC-20. The NeoRust SDK provides a convenient way to work with NEP-17 tokens:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account
    let account = wallet.default_account()?;

    // NEP-17 token contract hash
    let token_hash =
"0x1234567890abcdef1234567890abcdef12345678".parse::
<ScriptHash>()?;

    // Create a NEP-17 token instance
    let token = Nep17Contract::new(token_hash,
provider.clone());

    // Get token information
    let symbol = token.symbol().await?;
    let decimals = token.decimals().await?;
    let total_supply = token.total_supply().await?;

    println!("Token: {} (Decimals: {})", symbol, decimals);
    println!("Total Supply: {}", total_supply);

    // Get account balance
    let balance = token.balance_of(account.address()).await?;
    println!("Your Balance: {}", balance);

    // Transfer tokens
    let recipient =
"NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAaj".parse::<Address>()?;
    let amount = 100;
    let txid = token.transfer(account, recipient, amount,
None).await?;
```

```
        println!("Transfer sent with transaction ID: {}", txid);

        Ok(())
    }
```

Contract Events

Neo smart contracts can emit events that you can listen for:

```rust
use neo::prelude::*;
use std::time::Duration;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node with WebSocket
support
    let provider =
Provider::new_ws("wss://testnet1.neo.coz.io:4443/ws").await?;

    // Contract script hash (address)
    let contract_hash =
"0x1234567890abcdef1234567890abcdef12345678".parse::
<ScriptHash>()?;

    // Subscribe to contract events
    let mut events =
provider.subscribe_contract_event(contract_hash).await?;

    println!("Listening for events from contract {}...",
contract_hash);

    // Process events as they arrive
    while let Some(event) = events.next().await {
        println!("Event received: {:?}", event);

        // Process specific event types
        if event.event_name == "Transfer" {
            if let Some(from) = event.state.get(0) {
                if let Some(to) = event.state.get(1) {
                    if let Some(amount) = event.state.get(2)
{
                        println!("Transfer: {} from {} to
{}",

amount.as_integer().unwrap_or_default(),
                            from.as_address().map(|a|
a.to_string()).unwrap_or_default(),
                            to.as_address().map(|a|
a.to_string()).unwrap_or_default()
                        );
                    }
                }
            }
        }
```

```
        }

        Ok(())
    }
```

Best Practices

1. **Test on TestNet First**: Always test your contract interactions on TestNet before moving to MainNet.
2. **Gas Estimation**: Use the `estimate_gas` method to estimate the gas cost of your transactions.
3. **Error Handling**: Implement proper error handling for contract interactions.
4. **Event Monitoring**: Set up event monitoring for important contract events.
5. **Security**: Carefully review contract code and parameters before deployment or interaction.

Transactions

This tutorial covers creating and sending transactions on the Neo N3 blockchain using the NeoRust SDK.

Understanding Neo Transactions

Neo N3 transactions are the fundamental units of work in the Neo blockchain. They represent operations such as token transfers, smart contract invocations, and more. Each transaction has the following key components:

- **Version**: The transaction format version
- **Nonce**: A random number to prevent replay attacks
- **Sender**: The account initiating the transaction
- **System Fee**: Fee for executing the transaction
- **Network Fee**: Fee for including the transaction in a block
- **Valid Until Block**: Block height until which the transaction is valid
- **Script**: The VM script to execute
- **Signers**: Accounts that need to sign the transaction
- **Witnesses**: Signatures and verification scripts

Creating a Basic Transaction

Here's how to create a basic transaction using the TransactionBuilder:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will send the transaction
    let account = wallet.default_account()?;

    // Create a transaction
    let transaction = TransactionBuilder::new()
        .version(0)
        .nonce(rand::random::<u32>())
        .valid_until_block(provider.get_block_count().await?
+ 100)
        .script(
            ScriptBuilder::new()
                .emit_app_call(

"d2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?,
                    "transfer",
                    &[

ContractParameter::hash160(account.address().script_hash()),

ContractParameter::hash160("NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAa
j".parse::<Address>()?),
                        ContractParameter::integer(1000),
                        ContractParameter::any(None),
                    ],
                )
                .to_array()
        )
        .sign(account)?
        .build();
```

```rust
    // Send the transaction
    let txid =
provider.send_raw_transaction(&transaction).await?;
    println!("Transaction sent with ID: {}", txid);

    // Wait for the transaction to be confirmed
    let receipt = provider.wait_for_transaction(&txid, 60,
2).await?;
    println!("Transaction confirmed: {:?}", receipt);

    Ok(())
}
```

Transferring NEO or GAS

The NeoRust SDK provides convenient methods for transferring NEO and GAS tokens:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will send the transaction
    let account = wallet.default_account()?;

    // Create a GAS token instance
    let gas_token = GasToken::new(provider.clone());

    // Transfer 1 GAS to another address
    let recipient =
"NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAaj".parse::<Address>()?;
    let amount = 1_00000000; // 1 GAS (with 8 decimals)

    let txid = gas_token.transfer(account, recipient, amount,
None).await?;
    println!("GAS transfer sent with transaction ID: {}",
txid);

    // Similarly for NEO token
    let neo_token = NeoToken::new(provider.clone());

    // Transfer 1 NEO to another address
    let neo_amount = 1_00000000; // 1 NEO (with 8 decimals)

    let neo_txid = neo_token.transfer(account, recipient,
neo_amount, None).await?;
    println!("NEO transfer sent with transaction ID: {}",
neo_txid);
```

```
        Ok(())
    }
```

Multi-signature Transactions

Neo supports multi-signature accounts, which require multiple signatures to authorize a transaction:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load wallets for all signers
    let wallet1_path = Path::new("wallet1.json");
    let wallet2_path = Path::new("wallet2.json");
    let wallet3_path = Path::new("wallet3.json");

    let password = "my-secure-password";

    let wallet1 = Wallet::load(wallet1_path, password)?;
    let wallet2 = Wallet::load(wallet2_path, password)?;
    let wallet3 = Wallet::load(wallet3_path, password)?;

    let account1 = wallet1.default_account()?;
    let account2 = wallet2.default_account()?;
    let account3 = wallet3.default_account()?;

    // Create a multi-signature account (2 of 3)
    let multi_sig_account = Account::create_multi_sig(
        2,
        &[
            account1.public_key().clone(),
            account2.public_key().clone(),
            account3.public_key().clone(),
        ],
    )?;

    println!("Multi-signature address: {}",
multi_sig_account.address());

    // Create a transaction from the multi-signature account
    let mut transaction = TransactionBuilder::new()
        .version(0)
        .nonce(rand::random::<u32>())
        .valid_until_block(provider.get_block_count().await?
+ 100)
        .script(
            ScriptBuilder::new()
```

```rust
                .emit_app_call(

"d2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?,
                    "transfer",
                    &[

ContractParameter::hash160(multi_sig_account.address().script
_hash()),

ContractParameter::hash160("NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAa
j".parse::<Address>()?),
                        ContractParameter::integer(1000),
                        ContractParameter::any(None),
                    ],
                )
                .to_array()
        )
        .build();

    // Sign with the required number of accounts (2 of 3)
    transaction = transaction.sign(account1)?;
    transaction = transaction.sign(account2)?;

    // Send the transaction
    let txid =
provider.send_raw_transaction(&transaction).await?;
    println!("Multi-signature transaction sent with ID: {}",
txid);

    Ok(())
}
```

Transaction Fees

Neo N3 transactions require two types of fees:

1. **System Fee**: Cost of executing the transaction script
2. **Network Fee**: Cost of including the transaction in a block

You can estimate these fees before sending a transaction:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will send the transaction
    let account = wallet.default_account()?;

    // Create a transaction
    let transaction = TransactionBuilder::new()
        .version(0)
        .nonce(rand::random::<u32>())
        .valid_until_block(provider.get_block_count().await?
+ 100)
        .script(
            ScriptBuilder::new()
                .emit_app_call(

"d2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?,
                    "transfer",
                    &[

ContractParameter::hash160(account.address().script_hash()),

ContractParameter::hash160("NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAa
j".parse::<Address>()?),
                        ContractParameter::integer(1000),
                        ContractParameter::any(None),
                    ],
                )
                .to_array()
        )
        .build();

    // Estimate system fee
```

```rust
    let system_fee =
provider.estimate_system_fee(&transaction).await?;
    println!("Estimated system fee: {} GAS", system_fee);

    // Estimate network fee
    let network_fee =
provider.estimate_network_fee(&transaction).await?;
    println!("Estimated network fee: {} GAS", network_fee);

    // Total fee
    let total_fee = system_fee + network_fee;
    println!("Total estimated fee: {} GAS", total_fee);

    // Add fees to the transaction
    let transaction_with_fees =
TransactionBuilder::from_transaction(transaction)
        .system_fee(system_fee)
        .network_fee(network_fee)
        .sign(account)?
        .build();

    // Send the transaction
    let txid =
provider.send_raw_transaction(&transaction_with_fees).await?;
    println!("Transaction sent with ID: {}", txid);

    Ok(())
}
```

Checking Transaction Status

You can check the status of a transaction after sending it:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Transaction ID to check
    let txid =
"0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890
abcdef".parse::<TxHash>()?;

    // Get transaction
    let transaction = provider.get_transaction(&txid).await?;

    if let Some(tx) = transaction {
        println!("Transaction found: {:?}", tx);

        // Get application log
        let app_log =
provider.get_application_log(&txid).await?;

        if let Some(log) = app_log {
            println!("Transaction execution:");
            println!("  VM State: {}",
log.execution.vm_state);
            println!("  Gas Consumed: {}",
log.execution.gas_consumed);

            for (i, notification) in
log.execution.notifications.iter().enumerate() {
                println!("  Notification #{}: {}", i + 1,
notification.event_name);
                println!("    Contract: {}",
notification.contract);
                println!("    State: {:?}",
notification.state);
            }
        }
    } else {
        println!("Transaction not found. It may be pending or
invalid.");
    }
```

```
        Ok(())
    }
```

Best Practices

1. **Always Verify Addresses**: Double-check recipient addresses before sending transactions.
2. **Set Appropriate Valid Until Block**: Set a reasonable expiration for your transactions.
3. **Estimate Fees**: Always estimate and include appropriate fees to ensure your transaction is processed.
4. **Wait for Confirmation**: Always wait for transaction confirmation before considering it complete.
5. **Error Handling**: Implement proper error handling for transaction failures.
6. **Test on TestNet**: Always test your transactions on TestNet before moving to MainNet.

NEP-17 Tokens

This tutorial covers working with NEP-17 tokens on the Neo N3 blockchain using the NeoRust SDK.

Understanding NEP-17

NEP-17 is Neo's token standard, similar to Ethereum's ERC-20. It defines a standard interface for fungible tokens on the Neo blockchain. NEP-17 tokens have the following key methods:

- `symbol` : Returns the token's symbol
- `decimals` : Returns the number of decimal places the token uses
- `totalSupply` : Returns the total token supply
- `balanceOf` : Returns the token balance of a specific address
- `transfer` : Transfers tokens from one address to another

Creating a NEP-17 Token Instance

To interact with a NEP-17 token, you first need to create a token instance:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // NEP-17 token contract hash (e.g., GAS token)
    let gas_hash =
"0xd2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?;

    // Create a NEP-17 token instance
    let gas_token = Nep17Contract::new(gas_hash,
provider.clone());

    Ok(())
}
```

Getting Token Information

You can retrieve basic information about a token:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // NEP-17 token contract hash (e.g., GAS token)
    let gas_hash =
"0xd2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?;

    // Create a NEP-17 token instance
    let gas_token = Nep17Contract::new(gas_hash,
provider.clone());

    // Get token information
    let symbol = gas_token.symbol().await?;
    let decimals = gas_token.decimals().await?;
    let total_supply = gas_token.total_supply().await?;

    println!("Token: {} (Decimals: {})", symbol, decimals);
    println!("Total Supply: {}", total_supply);

    Ok(())
}
```

Checking Token Balance

You can check the token balance of an address:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // NEP-17 token contract hash (e.g., GAS token)
    let gas_hash =
"0xd2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?;

    // Create a NEP-17 token instance
    let gas_token = Nep17Contract::new(gas_hash,
provider.clone());

    // Address to check balance for
    let address =
"NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAaj".parse::<Address>()?;

    // Get token balance
    let balance = gas_token.balance_of(address).await?;

    println!("Balance: {}", balance);

    // For better display, consider the token's decimals
    let decimals = gas_token.decimals().await?;
    let formatted_balance = balance as f64 /
10f64.powi(decimals as i32);

    println!("Formatted Balance: {} {}", formatted_balance,
gas_token.symbol().await?);

    Ok(())
}
```

Transferring Tokens

You can transfer tokens from one address to another:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will send the tokens
    let account = wallet.default_account()?;

    // NEP-17 token contract hash (e.g., GAS token)
    let gas_hash =
"0xd2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?;

    // Create a NEP-17 token instance
    let gas_token = Nep17Contract::new(gas_hash,
provider.clone());

    // Recipient address
    let recipient =
"NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAaj".parse::<Address>()?;

    // Amount to transfer (considering decimals)
    let decimals = gas_token.decimals().await?;
    let amount = 1 * 10i64.pow(decimals as u32); // 1 token
with proper decimal places

    // Transfer tokens
    let txid = gas_token.transfer(account, recipient, amount,
None).await?;

    println!("Transfer sent with transaction ID: {}", txid);

    // Wait for the transaction to be confirmed
    let receipt = provider.wait_for_transaction(&txid, 60,
2).await?;
    println!("Transaction confirmed: {:?}", receipt);
```

```
        Ok(())
    }
```

Working with NEO and GAS Tokens

The NeoRust SDK provides specialized classes for the native NEO and GAS tokens:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account
    let account = wallet.default_account()?;

    // Create NEO token instance
    let neo_token = NeoToken::new(provider.clone());

    // Create GAS token instance
    let gas_token = GasToken::new(provider.clone());

    // Get NEO balance
    let neo_balance =
neo_token.balance_of(account.address()).await?;
    println!("NEO Balance: {}", neo_balance);

    // Get GAS balance
    let gas_balance =
gas_token.balance_of(account.address()).await?;
    println!("GAS Balance: {}", gas_balance);

    // Transfer NEO
    let recipient =
"NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAaj".parse::<Address>()?;
    let neo_amount = 1_00000000; // 1 NEO (with 8 decimals)

    let neo_txid = neo_token.transfer(account, recipient,
neo_amount, None).await?;
    println!("NEO transfer sent with transaction ID: {}",
neo_txid);

    // Transfer GAS
    let gas_amount = 1_00000000; // 1 GAS (with 8 decimals)
```

```rust
    let gas_txid = gas_token.transfer(account, recipient,
gas_amount, None).await?;
    println!("GAS transfer sent with transaction ID: {}",
gas_txid);

    Ok(())
}
```

Monitoring Token Transfers

You can monitor token transfers by subscribing to the Transfer event:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node with WebSocket
support
    let provider =
Provider::new_ws("wss://testnet1.neo.coz.io:4443/ws").await?;

    // NEP-17 token contract hash (e.g., GAS token)
    let gas_hash =
"0xd2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?;

    // Subscribe to Transfer events
    let mut events =
provider.subscribe_contract_event(gas_hash).await?;

    println!("Listening for token transfers...");

    // Process events as they arrive
    while let Some(event) = events.next().await {
        if event.event_name == "Transfer" {
            if let Some(from) = event.state.get(0) {
                if let Some(to) = event.state.get(1) {
                    if let Some(amount) = event.state.get(2)
{
                        println!("Transfer: {} tokens from {}
to {}",

amount.as_integer().unwrap_or_default(),
                            from.as_address().map(|a|
a.to_string()).unwrap_or_default(),
                            to.as_address().map(|a|
a.to_string()).unwrap_or_default()
                        );
                    }
                }
            }
        }
    }
```

```
        Ok(())
    }
```

Working with Famous Neo N3 Contracts

The NeoRust SDK provides direct support for several famous Neo N3 contracts:

Flamingo Finance

```rust
use neo::prelude::*;
use neo::neo_contract::famous::flamingo::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 MainNet node
    let provider =
Provider::new_http("https://mainnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account
    let account = wallet.default_account()?;

    // Create Flamingo Finance instance
    let flamingo = FlamingoFinance::new(provider.clone());

    // Get FLM token balance
    let flm_balance =
flamingo.flm_token().balance_of(account.address()).await?;
    println!("FLM Balance: {}", flm_balance);

    // Get liquidity pool information
    let pool_info =
flamingo.get_pool_info(FlamingoPool::NeoGas).await?;
    println!("Pool Info: {:?}", pool_info);

    // Add liquidity to a pool
    let neo_amount = 1_00000000; // 1 NEO
    let gas_amount = 1_00000000; // 1 GAS

    let add_liquidity_txid = flamingo.add_liquidity(
        account,
        FlamingoPool::NeoGas,
        neo_amount,
        gas_amount,
        None,
    ).await?;
```

```rust
    println!("Add Liquidity transaction ID: {}",
add_liquidity_txid);

    Ok(())
}
```

NeoburgerNeo

```rust
use neo::prelude::*;
use neo::neo_contract::famous::neoburger::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 MainNet node
    let provider =
Provider::new_http("https://mainnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account
    let account = wallet.default_account()?;

    // Create NeoburgerNeo instance
    let neoburger = NeoburgerNeo::new(provider.clone());

    // Get bNEO token balance
    let bneo_balance =
neoburger.bneo_token().balance_of(account.address()).await?;
    println!("bNEO Balance: {}", bneo_balance);

    // Wrap NEO to get bNEO
    let neo_amount = 1_00000000; // 1 NEO
    let wrap_txid = neoburger.wrap_neo(account,
neo_amount).await?;
    println!("Wrap NEO transaction ID: {}", wrap_txid);

    // Unwrap bNEO to get NEO
    let bneo_amount = 1_00000000; // 1 bNEO
    let unwrap_txid = neoburger.unwrap_bneo(account,
bneo_amount).await?;
    println!("Unwrap bNEO transaction ID: {}", unwrap_txid);

    Ok(())
}
```

Best Practices

1. **Check Balances Before Transfers**: Always check that an account has sufficient balance before attempting a transfer.
2. **Consider Decimals**: Remember to account for token decimals when displaying balances or specifying transfer amounts.
3. **Wait for Confirmations**: Always wait for transaction confirmations before considering a transfer complete.
4. **Error Handling**: Implement proper error handling for token operations.
5. **Gas Costs**: Be aware of the gas costs associated with token transfers and other operations.
6. **Test on TestNet**: Always test your token operations on TestNet before moving to MainNet.

Neo Name Service (NNS)

This tutorial covers working with the Neo Name Service (NNS) on the Neo N3 blockchain using the NeoRust SDK.

Understanding NNS

The Neo Name Service (NNS) is a distributed, open-source naming system based on the Neo blockchain. It maps human-readable names to machine-readable identifiers such as Neo addresses, contract script hashes, and more. This makes it easier to work with blockchain addresses and resources.

Key Concepts

- **Domain**: A human-readable name registered in the NNS (e.g., `example.neo`)
- **Record**: Data associated with a domain (e.g., address, text record, etc.)
- **TTL**: Time-to-live for a domain record
- **Owner**: The account that owns a domain and can manage its records
- **Resolver**: Contract that translates between domain names and addresses/resources

Creating an NNS Instance

To interact with the NNS, you first need to create an NNS instance:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Create an NNS instance
    let nns = NameService::new(provider.clone());

    Ok(())
}
```

Checking Domain Availability

Before registering a domain, you should check if it's available:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Create an NNS instance
    let nns = NameService::new(provider.clone());

    // Check if a domain is available
    let domain = "example.neo";
    let is_available = nns.is_available(domain).await?;

    if is_available {
        println!("Domain {} is available for registration",
domain);
    } else {
        println!("Domain {} is already registered", domain);
    }

    Ok(())
}
```

Registering a Domain

If a domain is available, you can register it:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will register the domain
    let account = wallet.default_account()?;

    // Create an NNS instance
    let nns = NameService::new(provider.clone());

    // Check if a domain is available
    let domain = "example.neo";
    let is_available = nns.is_available(domain).await?;

    if is_available {
        // Register the domain
        let registration_period = 1; // in years
        let txid = nns.register(account, domain,
registration_period).await?;

        println!("Domain registration initiated with
transaction ID: {}", txid);

        // Wait for the transaction to be confirmed
        let receipt = provider.wait_for_transaction(&txid,
60, 2).await?;
        println!("Domain registration confirmed: {:?}",
receipt);
    } else {
        println!("Domain {} is already registered", domain);
    }
```

```
        Ok(())
    }
```

Setting Domain Records

Once you own a domain, you can set various records for it:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that owns the domain
    let account = wallet.default_account()?;

    // Create an NNS instance
    let nns = NameService::new(provider.clone());

    // Domain name
    let domain = "example.neo";

    // Set an address record
    let address = account.address();
    let txid = nns.set_address(account, domain,
address).await?;

    println!("Address record set with transaction ID: {}",
txid);

    // Wait for the transaction to be confirmed
    let receipt = provider.wait_for_transaction(&txid, 60,
2).await?;
    println!("Address record confirmed: {:?}", receipt);

    // Set a text record
    let key = "email";
    let value = "contact@example.neo";
    let text_txid = nns.set_text(account, domain, key,
value).await?;

    println!("Text record set with transaction ID: {}",
text_txid);
```

```
        Ok(())
    }
```

Resolving Domain Records

You can resolve domain records to get the associated data:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Create an NNS instance
    let nns = NameService::new(provider.clone());

    // Domain name
    let domain = "example.neo";

    // Resolve address
    let address = nns.resolve_address(domain).await?;

    if let Some(addr) = address {
        println!("Domain {} resolves to address: {}", domain,
addr);
    } else {
        println!("No address record found for domain {}",
domain);
    }

    // Resolve text record
    let key = "email";
    let text = nns.resolve_text(domain, key).await?;

    if let Some(value) = text {
        println!("Text record '{}' for domain {}: {}", key,
domain, value);
    } else {
        println!("No text record '{}' found for domain {}",
key, domain);
    }

    Ok(())
}
```

Renewing a Domain

Domains need to be renewed periodically to maintain ownership:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that owns the domain
    let account = wallet.default_account()?;

    // Create an NNS instance
    let nns = NameService::new(provider.clone());

    // Domain name
    let domain = "example.neo";

    // Check domain expiration
    let expiration = nns.get_expiration(domain).await?;

    if let Some(exp) = expiration {
        println!("Domain {} expires at: {}", domain, exp);

        // Renew the domain
        let renewal_period = 1; // in years
        let txid = nns.renew(account, domain,
renewal_period).await?;

        println!("Domain renewal initiated with transaction
ID: {}", txid);

        // Wait for the transaction to be confirmed
        let receipt = provider.wait_for_transaction(&txid,
60, 2).await?;
        println!("Domain renewal confirmed: {:?}", receipt);
    } else {
        println!("Domain {} is not registered", domain);
    }
```

```
        Ok(())
    }
```

Transferring Domain Ownership

You can transfer ownership of a domain to another address:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that owns the domain
    let account = wallet.default_account()?;

    // Create an NNS instance
    let nns = NameService::new(provider.clone());

    // Domain name
    let domain = "example.neo";

    // New owner address
    let new_owner =
"NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAaj".parse::<Address>()?;

    // Transfer ownership
    let txid = nns.transfer(account, domain,
new_owner).await?;

    println!("Domain transfer initiated with transaction ID:
{}", txid);

    // Wait for the transaction to be confirmed
    let receipt = provider.wait_for_transaction(&txid, 60,
2).await?;
    println!("Domain transfer confirmed: {:?}", receipt);

    Ok(())
}
```

Using NNS in Applications

You can integrate NNS resolution into your applications to allow users to use domain names instead of addresses:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will send tokens
    let account = wallet.default_account()?;

    // Create an NNS instance
    let nns = NameService::new(provider.clone());

    // Create a GAS token instance
    let gas_token = GasToken::new(provider.clone());

    // Domain or address input from user
    let recipient_input = "example.neo";

    // Determine if input is a domain or address
    let recipient_address = if
recipient_input.ends_with(".neo") {
        // Resolve domain to address
        match nns.resolve_address(recipient_input).await? {
            Some(addr) => addr,
            None => {
                println!("Could not resolve domain {}",
recipient_input);
                return Ok(());
            }
        }
    } else {
        // Parse as address directly
        recipient_input.parse::<Address>()?
    };

    // Amount to transfer
    let amount = 1_00000000; // 1 GAS (with 8 decimals)
```

```
    // Transfer GAS
    let txid = gas_token.transfer(account, recipient_address,
amount, None).await?;
    println!("Transfer sent to {} with transaction ID: {}",
recipient_input, txid);

    Ok(())
}
```

Best Practices

1. **Check Domain Availability**: Always check if a domain is available before attempting to register it.
2. **Monitor Expiration**: Keep track of domain expiration dates and renew domains before they expire.
3. **Secure Ownership**: Ensure that the account owning valuable domains is properly secured.
4. **Validate Input**: When accepting domain names as input, validate them before attempting to resolve.
5. **Handle Resolution Failures**: Always handle cases where domain resolution fails gracefully.
6. **Test on TestNet**: Always test your NNS operations on TestNet before moving to MainNet.

Neo X Integration

This tutorial covers working with Neo X, an EVM-compatible chain maintained by Neo, using the NeoRust SDK.

Understanding Neo X

Neo X is an EVM-compatible chain maintained by the Neo ecosystem. It provides Ethereum compatibility while leveraging Neo's infrastructure and security. Key features include:

- **EVM Compatibility**: Run Ethereum smart contracts and use Ethereum tools
- **Bridge Functionality**: Transfer tokens between Neo N3 and Neo X
- **Shared Security**: Benefit from Neo's consensus mechanism
- **Cross-Chain Interoperability**: Interact with both Neo and Ethereum ecosystems

Setting Up Neo X Provider

To interact with Neo X, you first need to create a Neo X provider:

```rust
use neo::prelude::*;
use neo::neo_x::evm::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo X node
    let provider =
NeoXProvider::new_http("https://rpc.neoX.io");

    // Get the current block number
    let block_number = provider.get_block_number().await?;
    println!("Current Neo X block number: {}", block_number);

    // Get chain ID
    let chain_id = provider.get_chain_id().await?;
    println!("Neo X chain ID: {}", chain_id);

    Ok(())
}
```

Creating and Sending Neo X Transactions

You can create and send transactions on the Neo X chain:

```rust
use neo::prelude::*;
use neo::neo_x::evm::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo X node
    let provider =
NeoXProvider::new_http("https://rpc.neoX.io");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will send the transaction
    let account = wallet.default_account()?;

    // Create a transaction
    let transaction = NeoXTransaction::new()
        .to("0x1234567890123456789012345678901234567890")
        .value(1_000_000_000_000_000_000u128) // 1 ETH in wei
        .gas_price(20_000_000_000u64) // 20 Gwei
        .gas_limit(21_000u64)

.nonce(provider.get_transaction_count(account.address().to_eth_address(), None).await?)
        .chain_id(provider.get_chain_id().await?)
        .build();

    // Sign the transaction
    let signed_tx = transaction.sign(account)?;

    // Send the transaction
    let txid =
provider.send_raw_transaction(&signed_tx).await?;
    println!("Transaction sent with ID: {}", txid);

    // Wait for the transaction to be confirmed
    let receipt = provider.wait_for_transaction(&txid, 60,
2).await?;
    println!("Transaction confirmed: {:?}", receipt);
```

```
        Ok(())
    }
```

Interacting with EVM Smart Contracts

You can interact with EVM smart contracts on Neo X:

```rust
use neo::prelude::*;
use neo::neo_x::evm::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo X node
    let provider =
NeoXProvider::new_http("https://rpc.neoX.io");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will interact with the contract
    let account = wallet.default_account()?;

    // ERC-20 token contract address
    let contract_address =
"0x1234567890123456789012345678901234567890";

    // Create a contract instance
    let contract = NeoXContract::new(contract_address,
provider.clone());

    // Call a read-only method (balanceOf)
    let balance = contract.call_read(
        "balanceOf",
        &[account.address().to_eth_address()],
    ).await?;

    println!("Token balance: {}",
balance.as_u256().unwrap_or_default());

    // Call a state-changing method (transfer)
    let recipient =
"0x0987654321098765432109876543210987654321";
    let amount = 1_000_000_000_000_000_000u128; // 1 token
with 18 decimals

    let tx = contract.call_write(
        account,
        "transfer",
        &[recipient, amount.to_string()],
```

```
            None,
    ).await?;

    println!("Transfer transaction sent with ID: {}", tx);

    Ok(())
}
```

Using the Neo X Bridge

The Neo X Bridge allows you to transfer tokens between Neo N3 and Neo X:

```rust
use neo::prelude::*;
use neo::neo_x::bridge::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to Neo N3 and Neo X nodes
    let neo_provider =
Provider::new_http("https://mainnet1.neo.coz.io:443");
    let neox_provider =
NeoXProvider::new_http("https://rpc.neoX.io");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account
    let account = wallet.default_account()?;

    // Create a bridge contract instance
    let bridge =
NeoXBridgeContract::new(neo_provider.clone(),
neox_provider.clone());

    // Bridge GAS from Neo N3 to Neo X
    let amount = 1_00000000; // 1 GAS (with 8 decimals)

    let txid = bridge.bridge_to_neox(
        account,
        BridgeToken::Gas,
        amount,
        account.address().to_eth_address(),
    ).await?;

    println!("Bridge transaction sent with ID: {}", txid);

    // Wait for the transaction to be confirmed and processed
by the bridge
    println!("Waiting for bridge processing (this may take
several minutes)...");
    let receipt = neo_provider.wait_for_transaction(&txid,
300, 2).await?;
    println!("Bridge transaction confirmed on Neo N3: {:?}",
receipt);
```

```
    // Check if tokens were received on Neo X
    // Note: There might be a delay before tokens appear on
Neo X
    let erc20_address =
bridge.get_neox_token_address(BridgeToken::Gas).await?;
    let contract = NeoXContract::new(erc20_address,
neox_provider.clone());

    let balance = contract.call_read(
        "balanceOf",
        &[account.address().to_eth_address()],
    ).await?;

    println!("Bridged GAS balance on Neo X: {}",
balance.as_u256().unwrap_or_default());

    Ok(())
}
```

Bridging Tokens from Neo X to Neo N3

You can also bridge tokens from Neo X back to Neo N3:

```rust
use neo::prelude::*;
use neo::neo_x::bridge::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to Neo N3 and Neo X nodes
    let neo_provider =
Provider::new_http("https://mainnet1.neo.coz.io:443");
    let neox_provider =
NeoXProvider::new_http("https://rpc.neoX.io");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account
    let account = wallet.default_account()?;

    // Create a bridge contract instance
    let bridge =
NeoXBridgeContract::new(neo_provider.clone(),
neox_provider.clone());

    // Bridge GAS from Neo X to Neo N3
    let amount = 1_000_000_000_000_000_000u128; // 1 GAS
(with 18 decimals on Neo X)

    let txid = bridge.bridge_to_neo(
        account,
        BridgeToken::Gas,
        amount,
        account.address(),
    ).await?;

    println!("Bridge transaction sent with ID: {}", txid);

    // Wait for the transaction to be confirmed and processed
by the bridge
    println!("Waiting for bridge processing (this may take
several minutes)...");
    let receipt = neox_provider.wait_for_transaction(&txid,
300, 2).await?;
    println!("Bridge transaction confirmed on Neo X: {:?}",
```

```
    receipt);

    // Check if tokens were received on Neo N3
    // Note: There might be a delay before tokens appear on
Neo N3
    let gas_token = GasToken::new(neo_provider.clone());
    let balance =
gas_token.balance_of(account.address()).await?;

    println!("GAS balance on Neo N3: {}", balance);

    Ok(())
}
```

Monitoring Bridge Events

You can monitor bridge events to track token transfers between chains:

```rust
use neo::prelude::*;
use neo::neo_x::bridge::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to Neo N3 and Neo X nodes with WebSocket
support
    let neo_provider =
Provider::new_ws("wss://mainnet1.neo.coz.io:4443/ws").await?;
    let neox_provider =
NeoXProvider::new_ws("wss://ws.neoX.io").await?;

    // Create a bridge contract instance
    let bridge =
NeoXBridgeContract::new(neo_provider.clone(),
neox_provider.clone());

    // Subscribe to bridge events on Neo N3
    let mut neo_events =
bridge.subscribe_neo_events().await?;

    println!("Listening for bridge events on Neo N3...");

    // Process Neo N3 events in a separate task
    tokio::spawn(async move {
        while let Some(event) = neo_events.next().await {
            println!("Neo N3 Bridge Event: {:?}", event);

            if event.event_name == "TokensLocked" {
                if let Some(from) = event.state.get(0) {
                    if let Some(to) = event.state.get(1) {
                        if let Some(amount) =
event.state.get(2) {
                            if let Some(token) =
event.state.get(3) {
                                println!("Tokens Locked: {}
{} from {} to {}",

amount.as_integer().unwrap_or_default(),

token.as_string().unwrap_or_default(),
                                    from.as_address().map(|a|
a.to_string()).unwrap_or_default(),

to.as_string().unwrap_or_default()
```

```rust
                                    );
                                }
                            }
                        }
                    }
                }
        });

        // Subscribe to bridge events on Neo X
        let mut neox_events =
    bridge.subscribe_neox_events().await?;

        println!("Listening for bridge events on Neo X...");

        // Process Neo X events
        while let Some(event) = neox_events.next().await {
            println!("Neo X Bridge Event: {:?}", event);

            if event.event_name == "TokensUnlocked" {
                println!("Tokens Unlocked: from {} to {} amount
{}",
                    event.get_param("from").unwrap_or_default(),
                    event.get_param("to").unwrap_or_default(),
                    event.get_param("amount").unwrap_or_default()
                );
            }
        }

        Ok(())
}
```

Best Practices

1. **Gas Management**: Be aware of gas costs on Neo X, which follow Ethereum's gas model.
2. **Bridge Delays**: Expect delays when bridging tokens between chains, as cross-chain operations require confirmations on both chains.
3. **Address Formats**: Remember that Neo N3 and Neo X use different address formats. Use the appropriate conversion methods.

4. **Security**: Always verify addresses and amounts before sending transactions or bridging tokens.
5. **Testing**: Test bridge operations with small amounts before transferring larger values.
6. **Error Handling**: Implement proper error handling for both Neo N3 and Neo X operations.
7. **Monitoring**: Set up monitoring for bridge events to track the status of cross-chain transfers.

SGX Support

This tutorial covers using the Intel SGX (Software Guard Extensions) features of the NeoRust SDK for secure blockchain operations.

Understanding Intel SGX

Intel SGX is a set of security-related instruction codes built into modern Intel CPUs. It allows user-level code to allocate private regions of memory, called enclaves, which are protected from processes running at higher privilege levels. Key benefits include:

- **Hardware-Level Security**: Protection of sensitive data and code from the operating system, hypervisor, BIOS, and other privileged software
- **Secure Computation**: Ability to perform computations on sensitive data within the enclave
- **Remote Attestation**: Verification that code is running in a genuine SGX enclave
- **Sealing**: Secure storage of sensitive data for later use within an enclave

Prerequisites

Before using the SGX features of NeoRust, you need:

1. **Intel SGX-compatible hardware**

    - CPU with SGX support (check with `cpuid` or Intel's processor list)
    - SGX enabled in BIOS/UEFI

2. **Intel SGX Software Stack**

    - Intel SGX Driver
    - Intel SGX SDK v2.12

- Intel SGX PSW (Platform Software)

3. **Rust Toolchain**

- Rust nightly-2022-10-22 (required by the Apache Teaclave SGX SDK)
- Install with: `rustup install nightly-2022-10-22`
- Set as default for this project: `rustup override set nightly-2022-10-22`

For detailed installation instructions, see the SGX Setup Guide.

Enabling SGX Support in NeoRust

To enable SGX support in your project, add the `sgx` feature to your Cargo.toml:

```
[dependencies]
neo = { git = "https://github.com/R3E-Network/NeoRust", features = ["sgx"] }
```

You also need to uncomment the SGX dependencies in the NeoRust Cargo.toml:

```
# SGX dependencies
sgx_types = { version = "=1.1.1", optional = true }
sgx_urts = { version = "=1.1.1", optional = true }
sgx_tstd = { version = "=1.1.1", optional = true }
sgx_tcrypto = { version = "=1.1.1", optional = true }

[features]
sgx = ["sgx-deps"]
sgx-deps = [
    "sgx_types",
    "sgx_urts",
    "sgx_tstd",
    "sgx_tcrypto"
]
```

Building the SGX Components

Use the provided Makefile for SGX to build the enclave components:

```
make -f Makefile.sgx
```

This will build both the trusted enclave components and the untrusted application components.

Creating an SGX Enclave Manager

The first step in using SGX features is to create an enclave manager:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Path to the enclave shared object
    let enclave_path = "path/to/enclave.so";

    // Initialize the SGX enclave
    let enclave_manager =
SgxEnclaveManager::new(enclave_path)?;
    println!("SGX enclave initialized successfully!");

    Ok(())
}
```

Secure Wallet Management with SGX

One of the primary use cases for SGX in blockchain applications is secure wallet management:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Path to the enclave shared object
    let enclave_path = "path/to/enclave.so";

    // Initialize the SGX enclave
    let enclave_manager =
SgxEnclaveManager::new(enclave_path)?;

    // Create a wallet with a password
    let password = "my-secure-password";
    let wallet = enclave_manager.create_wallet(password)?;

    // Get the wallet's public key
    let public_key = wallet.get_public_key();
    println!("Wallet public key: {:?}", public_key);

    // The private key never leaves the enclave

    Ok(())
}
```

Signing Transactions Securely

With SGX, you can sign transactions without exposing private keys:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Path to the enclave shared object
    let enclave_path = "path/to/enclave.so";

    // Initialize the SGX enclave
    let enclave_manager =
SgxEnclaveManager::new(enclave_path)?;

    // Create a wallet with a password
    let password = "my-secure-password";
    let wallet = enclave_manager.create_wallet(password)?;

    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Create a transaction
    let transaction = TransactionBuilder::new()
        .version(0)
        .nonce(rand::random::<u32>())
        .valid_until_block(provider.get_block_count().await?
+ 100)
        .script(
            ScriptBuilder::new()
                .contract_call(

"d2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?,
                    "transfer",
                    &[

ContractParameter::hash160(wallet.get_address().script_hash()
),

ContractParameter::hash160("NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAa
j".parse::<Address>()?),

ContractParameter::integer(1_00000000), // 1 GAS
                        ContractParameter::any(None),
                    ],
                )
```

```
            .to_array()
        )
        .build();

    // Sign the transaction securely within the enclave
    let signed_tx = wallet.sign_transaction(&transaction)?;

    // Send the transaction
    let txid =
provider.send_raw_transaction(&signed_tx).await?;
    println!("Transaction sent with ID: {}", txid);

    Ok(())
}
```

Secure RPC Client

The SGX module also provides a secure RPC client for blockchain interactions:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Path to the enclave shared object
    let enclave_path = "path/to/enclave.so";

    // Initialize the SGX enclave
    let enclave_manager =
SgxEnclaveManager::new(enclave_path)?;

    // Create a secure RPC client
    let rpc_url = "https://testnet1.neo.coz.io:443";
    let rpc_client =
enclave_manager.create_rpc_client(rpc_url)?;

    // Use the secure RPC client
    let block_count = rpc_client.get_block_count().await?;
    println!("Current block height: {}", block_count);

    // The RPC client encrypts sensitive data and performs
secure validation
    // of responses within the enclave

    Ok(())
}
```

Secure Storage

The SGX module provides secure storage for sensitive data:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Path to the enclave shared object
    let enclave_path = "path/to/enclave.so";

    // Initialize the SGX enclave
    let enclave_manager =
SgxEnclaveManager::new(enclave_path)?;

    // Create a secure storage instance
    let storage = enclave_manager.create_storage()?;

    // Store sensitive data
    let key = "api_key";
    let value = "my-secret-api-key";
    storage.set(key, value)?;

    // Retrieve sensitive data
    let retrieved_value = storage.get(key)?;
    println!("Retrieved value: {}", retrieved_value);

    // The data is encrypted and stored securely

    Ok(())
}
```

Remote Attestation

Remote attestation allows you to verify that your code is running in a genuine
SGX enclave:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Path to the enclave shared object
    let enclave_path = "path/to/enclave.so";

    // Initialize the SGX enclave
    let enclave_manager =
SgxEnclaveManager::new(enclave_path)?;

    // Generate a remote attestation quote
    let quote =
enclave_manager.generate_attestation_quote()?;

    // Send the quote to a remote verifier
    // (implementation depends on your attestation service)
    let attestation_service_url =
"https://attestation.example.com";
    let verification_result =
verify_quote_with_service(attestation_service_url,
&quote).await?;

    if verification_result.is_valid {
        println!("Remote attestation successful!");
        // Proceed with secure operations
    } else {
        println!("Remote attestation failed!");
        // Handle the failure
    }

    Ok(())
}

async fn verify_quote_with_service(url: &str, quote: &[u8]) -
> Result<VerificationResult, Box<dyn std::error::Error>> {
    // Implementation of quote verification with a remote
attestation service
    // This is just a placeholder
    Ok(VerificationResult { is_valid: true })
}

struct VerificationResult {
```

```
        is_valid: bool,
    }
```

Simulation Mode

If you don't have SGX hardware, you can still develop and test using simulation mode:

```
export SGX_MODE=SIM
make -f Makefile.sgx
```

In your code, you can check if you're running in simulation mode:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Path to the enclave shared object
    let enclave_path = "path/to/enclave.so";

    // Initialize the SGX enclave
    let enclave_manager =
SgxEnclaveManager::new(enclave_path)?;

    if enclave_manager.is_simulation_mode() {
        println!("Running in simulation mode. Security
guarantees are not provided.");
    } else {
        println!("Running in hardware mode with full SGX
protection.");
    }

    Ok(())
}
```

Best Practices

1. **Minimize Enclave Code**: Keep the enclave code small to reduce the attack surface.
2. **Validate Inputs**: Always validate inputs before passing them to the enclave.
3. **Secure Key Management**: Never extract private keys from the enclave.
4. **Use Remote Attestation**: Verify the integrity of the enclave in production environments.
5. **Regular Updates**: Keep the SGX SDK and drivers updated to address security vulnerabilities.
6. **Error Handling**: Implement proper error handling for enclave operations.
7. **Testing**: Test your SGX code in both simulation and hardware modes.

Security Considerations

When using SGX for blockchain applications:

1. **Side-Channel Attacks**: Be aware that SGX is not immune to all side-channel attacks.
2. **Enclave Interface**: The interface between the untrusted application and the enclave is a potential attack vector.
3. **Data Sealing**: Use data sealing to protect sensitive data at rest.
4. **Memory Limitations**: SGX enclaves have memory limitations; design your application accordingly.
5. **Attestation**: Use remote attestation in production to verify enclave integrity.

API Overview

This reference provides an overview of the NeoRust SDK API, including the main modules and their functionality.

Core Modules

The NeoRust SDK is organized into several core modules, each responsible for a specific aspect of Neo blockchain interaction:

neo_wallets

The `neo_wallets` module provides functionality for creating, loading, and managing Neo wallets and accounts.

```rust
use neo::prelude::*;

// Create a new wallet
let wallet = Wallet::new("password")?;

// Create a new account
let account = wallet.create_account()?;

// Get account address
let address = account.address();
```

Key components:

- `Wallet`: Manages multiple accounts and provides wallet-level operations
- `Account`: Represents a Neo account with a key pair
- `Address`: Represents a Neo address

neo_clients

The `neo_clients` module provides clients for interacting with Neo nodes via RPC.

```rust
use neo::prelude::*;

// Create a provider connected to a Neo node
let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

// Get the current block count
let block_count = provider.get_block_count().await?;
```

Key components:

- `Provider` : Main client for interacting with Neo nodes
- `RpcClient` : Low-level RPC client
- `WebSocketProvider` : Provider with WebSocket support for subscriptions

neo_types

The `neo_types` module provides fundamental Neo blockchain types.

```rust
use neo::prelude::*;

// Create a script hash from a string
let script_hash =
"0xd2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?;

// Create an address from a string
let address = "NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAaj".parse::
<Address>()?;

// Create a transaction hash from a string
let tx_hash =
"0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890
abcdef".parse::<TxHash>()?;
```

Key components:

- `Address` : Neo address
- `ScriptHash` : Contract script hash
- `TxHash` : Transaction hash
- `ContractParameter` : Parameter for contract invocation

neo_crypto

The `neo_crypto` module provides cryptographic functionality.

```
use neo::prelude::*;

// Generate a new key pair
let key_pair = KeyPair::new()?;

// Sign a message
let message = b"Hello, Neo!";
let signature = key_pair.sign_message(message)?;

// Verify a signature
let is_valid = key_pair.verify_signature(message,
&signature)?;
```

Key components:

- `KeyPair` : Represents a public/private key pair
- `PublicKey` : Represents a public key
- `PrivateKey` : Represents a private key
- `Signature` : Represents a cryptographic signature

neo_builder

The `neo_builder` module provides builders for creating transactions and scripts.

```rust
use neo::prelude::*;

// Create a transaction
let transaction = TransactionBuilder::new()
    .version(0)
    .nonce(rand::random::<u32>())
    .valid_until_block(block_count + 100)
    .script(script)
    .sign(account)?
    .build();

// Create a script
let script = ScriptBuilder::new()
    .contract_call(
        script_hash,
        "transfer",
        &[

ContractParameter::hash160(from_address.script_hash()),

ContractParameter::hash160(to_address.script_hash()),
            ContractParameter::integer(amount),
            ContractParameter::any(None),
        ],
    )
    .to_array();
```

Key components:

- `TransactionBuilder` : Builder for creating transactions
- `ScriptBuilder` : Builder for creating VM scripts

neo_contract

The `neo_contract` module provides interfaces for interacting with Neo smart contracts.

```rust
use neo::prelude::*;

// Create a NEP-17 token instance
let token = Nep17Contract::new(token_hash, provider.clone());

// Get token information
let symbol = token.symbol().await?;
let decimals = token.decimals().await?;
let total_supply = token.total_supply().await?;

// Get token balance
let balance = token.balance_of(address).await?;
```

Key components:

- `Nep17Contract` : Interface for NEP-17 tokens
- `NeoToken` : Interface for the NEO token
- `GasToken` : Interface for the GAS token
- `NameService` : Interface for the Neo Name Service

neo_x

The `neo_x` module provides support for Neo X, an EVM-compatible chain maintained by Neo.

```rust
use neo::prelude::*;
use neo::neo_x::evm::*;

// Create a Neo X provider
let provider = NeoXProvider::new_http("https://rpc.neoX.io");

// Create a transaction
let transaction = NeoXTransaction::new()
    .to("0x1234567890123456789012345678901234567890")
    .value(1_000_000_000_000_000_000u128) // 1 ETH in wei
    .gas_price(20_000_000_000u64) // 20 Gwei
    .gas_limit(21_000u64)
    .build();
```

Key components:

- `NeoXProvider` : Provider for interacting with Neo X nodes
- `NeoXTransaction` : Transaction for Neo X
- `NeoXBridgeContract` : Interface for the Neo X bridge

neo_sgx

The `neo_sgx` module provides support for Intel SGX (Software Guard Extensions) for secure operations.

```rust
use neo::prelude::*;

// Initialize the SGX enclave
let enclave_manager =
SgxEnclaveManager::new("path/to/enclave.so")?;

// Create a wallet with a password
let wallet = enclave_manager.create_wallet("password")?;

// Sign a transaction securely within the enclave
let signed_tx = wallet.sign_transaction(&transaction)?;
```

Key components:

- `SgxEnclaveManager` : Manager for SGX enclaves
- `SgxWallet` : Secure wallet implementation
- `SgxRpcClient` : Secure RPC client

Prelude

The `prelude` module re-exports commonly used types and functions for convenience:

```rust
use neo::prelude::*;
```

This imports all the essential types and functions you need for most operations with the NeoRust SDK.

Feature Flags

The NeoRust SDK supports various feature flags to enable specific functionality:

- `ledger` : Support for Ledger hardware wallets
- `aws` : AWS integration
- `sgx` : Intel SGX support

Enable these features in your Cargo.toml:

```toml
[dependencies]
neo = { git = "https://github.com/R3E-Network/NeoRust", features = ["ledger", "aws", "sgx"] }
```

Error Handling

The NeoRust SDK uses Rust's `Result` type for error handling. Most functions return a `Result<T, Error>` where `Error` is a custom error type that can represent various error conditions.

```rust
use neo::prelude::*;

fn example() -> Result<(), Box<dyn std::error::Error>> {
    // Create a provider
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Get the current block count
    match provider.get_block_count().await {
        Ok(block_count) => println!("Current block count:
{}", block_count),
        Err(e) => println!("Error: {}", e),
    }

    Ok(())
}
```

For more detailed information on specific modules and types, see the corresponding reference pages.

Configuration

This reference provides information about configuring the NeoRust SDK, including environment variables, network settings, and other configuration options.

Network Configuration

The NeoRust SDK supports connecting to different Neo networks, including MainNet, TestNet, and private networks.

Predefined Networks

The SDK includes predefined configurations for common Neo networks:

```rust
use neo::prelude::*;

// Connect to Neo N3 MainNet
let mainnet_provider =
Provider::new_http("https://mainnet1.neo.coz.io:443");

// Connect to Neo N3 TestNet
let testnet_provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

// Connect to a local Neo Express instance
let local_provider =
Provider::new_http("http://localhost:10332");
```

Custom Networks

You can also connect to custom Neo networks by providing the RPC URL:

```rust
use neo::prelude::*;

// Connect to a custom Neo N3 node
let custom_provider = Provider::new_http("https://my-custom-
neo-node.example.com:10332");
```

WebSocket Connections

For applications that need real-time updates, you can use WebSocket
connections:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node with WebSocket
support
    let ws_provider =
Provider::new_ws("wss://testnet1.neo.coz.io:4443/ws").await?;

    // Subscribe to new blocks
    let mut blocks = ws_provider.subscribe_blocks().await?;

    println!("Listening for new blocks...");

    // Process new blocks as they arrive
    while let Some(block) = blocks.next().await {
        println!("New block: {} (hash: {})", block.index,
block.hash);
    }

    Ok(())
}
```

SDK Configuration

The NeoRust SDK can be configured using the `Config` struct:

```rust
use neo::prelude::*;

// Create a custom configuration
let config = Config::new()
    .network(Network::TestNet)
    .timeout(std::time::Duration::from_secs(30))
    .max_retry(3)
    .build();

// Create a provider with the custom configuration
let provider =
Provider::with_config("https://testnet1.neo.coz.io:443",
config);
```

Configuration Options

The following options can be configured:

Option	Description	Default
`network`	The Neo network to connect to	`Network::MainNet`
`timeout`	Request timeout	30 seconds
`max_retry`	Maximum number of retry attempts	3
`retry_delay`	Delay between retry attempts	1 second
`user_agent`	User agent string for HTTP requests	`"NeoRust/{version}"`

Environment Variables

The NeoRust SDK respects the following environment variables:

Variable	Description	Default
`NEO_RPC_URL`	Default RPC URL for Neo N3	None
`NEO_WS_URL`	Default WebSocket URL for Neo N3	None
`NEO_NETWORK`	Default network (`mainnet`, `testnet`)	`mainnet`
`NEO_PRIVATE_KEY`	Default private key for signing transactions	None
`NEO_GAS_PRICE`	Default gas price for transactions	Network default
`NEO_LOG_LEVEL`	Logging level (`error`, `warn`, `info`, `debug`, `trace`)	`info`

You can set these environment variables in your shell or use a `.env` file with the `dotenv` crate:

```rust
use dotenv::dotenv;
use neo::prelude::*;

fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Load environment variables from .env file
    dotenv().ok();

    // Create a provider using the NEO_RPC_URL environment
variable
    let provider = Provider::from_env()?;

    Ok(())
}
```

Logging Configuration

The NeoRust SDK uses the `tracing` crate for logging. You can configure the logging level and output:

```rust
use neo::prelude::*;
use tracing_subscriber::{fmt, EnvFilter};

fn main() {
    // Initialize the logger with custom configuration
    tracing_subscriber::fmt()
        .with_env_filter(EnvFilter::from_default_env()
            .add_directive("neo=debug".parse().unwrap())
            .add_directive("warn".parse().unwrap()))
        .init();

    // Now logs will be output according to the configuration
}
```

You can also use the `NEO_LOG_LEVEL` environment variable to control the logging level:

```bash
# Set the log level to debug
export NEO_LOG_LEVEL=debug

# Run your application
cargo run
```

Gas Configuration

You can configure gas settings for transactions:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will send the transaction
    let account = wallet.default_account()?;

    // Create a transaction with custom gas settings
    let transaction = TransactionBuilder::new()
        .version(0)
        .nonce(rand::random::<u32>())
        .valid_until_block(provider.get_block_count().await?
+ 100)
        .script(
            ScriptBuilder::new()
                .contract_call(

"d2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?,
                    "transfer",
                    &[

ContractParameter::hash160(account.address().script_hash()),

ContractParameter::hash160("NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAa
j".parse::<Address>()?),

ContractParameter::integer(1_00000000), // 1 GAS
                        ContractParameter::any(None),
                    ],
                )
                .to_array()
        )
        .system_fee(1_00000000) // 1 GAS system fee
        .network_fee(0_50000000) // 0.5 GAS network fee
```

```rust
        .sign(account)?
        .build();

    // Send the transaction
    let txid =
provider.send_raw_transaction(&transaction).await?;
    println!("Transaction sent with ID: {}", txid);

    Ok(())
}
```

SGX Configuration

If you're using the SGX features, you can configure the SGX environment:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Configure SGX
    let sgx_config = SgxConfig::new()
        .enclave_path("path/to/enclave.so")
        .simulation_mode(false)
        .build();

    // Initialize the SGX enclave with the configuration
    let enclave_manager =
SgxEnclaveManager::with_config(sgx_config)?;

    // Use the enclave manager
    let wallet = enclave_manager.create_wallet("my-secure-
password")?;

    Ok(())
}
```

You can also use environment variables for SGX configuration:

Variable	Description	Default
`SGX_MODE`	SGX mode (`HW` or `SIM`)	`HW`
`SGX_ENCLAVE_PATH`	Path to the enclave shared object	None
`SGX_AESM_ADDR`	Address of the AESM service	`127.0.0.1:2222`

Best Practices

1. **Environment-Specific Configuration**: Use different configurations for development, testing, and production environments.
2. **Secure Credential Management**: Never hardcode private keys or passwords in your code.
3. **Timeout Configuration**: Set appropriate timeouts based on your network conditions.
4. **Logging Configuration**: Configure logging appropriately for your environment.
5. **Gas Estimation**: Use gas estimation functions instead of hardcoding gas values.
6. **Error Handling**: Implement proper error handling for configuration errors.

Error Handling

This reference provides information about error handling in the NeoRust SDK, including error types, error propagation, and best practices for handling errors.

Error Types

The NeoRust SDK uses a comprehensive error handling system based on Rust's `Result` type. The main error types in the SDK include:

NeoError

The `NeoError` is the primary error type used throughout the SDK. It encompasses various error categories:

```rust
pub enum NeoError {
    // RPC errors
    RpcError(RpcError),

    // Wallet errors
    WalletError(WalletError),

    // Cryptographic errors
    CryptoError(CryptoError),

    // Transaction errors
    TransactionError(TransactionError),

    // Contract errors
    ContractError(ContractError),

    // Serialization errors
    SerializationError(SerializationError),

    // IO errors
    IoError(std::io::Error),

    // Other errors
    Other(String),
}
```

RpcError

The `RpcError` represents errors that occur during RPC communication with
Neo nodes:

```rust
pub enum RpcError {
    // HTTP errors
    HttpError(reqwest::Error),

    // JSON-RPC errors
    JsonRpcError {
        code: i64,
        message: String,
        data: Option<serde_json::Value>,
    },

    // WebSocket errors
    WebSocketError(String),

    // Timeout errors
    TimeoutError,

    // Other errors
    Other(String),
}
```

WalletError

The `WalletError` represents errors related to wallet operations:

```rust
pub enum WalletError {
    // Password errors
    InvalidPassword,

    // Account errors
    AccountNotFound,
    InvalidAccount,

    // Key errors
    InvalidPrivateKey,
    InvalidPublicKey,

    // File errors
    FileError(std::io::Error),

    // Other errors
    Other(String),
}
```

CryptoError

The `CryptoError` represents errors related to cryptographic operations:

```rust
pub enum CryptoError {
    // Signature errors
    SignatureError,
    VerificationError,

    // Key errors
    InvalidKey,

    // Hash errors
    HashError,

    // Other errors
    Other(String),
}
```

TransactionError

The `TransactionError` represents errors related to transaction operations:

```rust
pub enum TransactionError {
    // Validation errors
    InvalidTransaction,
    InvalidSignature,

    // Fee errors
    InsufficientFunds,

    // Network errors
    NetworkError,

    // Other errors
    Other(String),
}
```

ContractError

The `ContractError` represents errors related to smart contract operations:

```rust
pub enum ContractError {
    // Invocation errors
    InvocationError,

    // Parameter errors
    InvalidParameter,

    // Execution errors
    ExecutionError,

    // Other errors
    Other(String),
}
```

Error Propagation

The NeoRust SDK uses Rust's `?` operator for error propagation. This allows for concise error handling code:

```rust
use neo::prelude::*;
use std::path::Path;

fn load_wallet_and_get_balance(
    wallet_path: &Path,
    password: &str,
    provider: &Provider,
    token_hash: ScriptHash,
) -> Result<u64, NeoError> {
    // Load the wallet
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the default account
    let account = wallet.default_account()?;

    // Create a NEP-17 token instance
    let token = Nep17Contract::new(token_hash,
provider.clone());

    // Get the token balance
    let balance = token.balance_of(account.address())?;

    Ok(balance)
}
```

In this example, if any of the operations fail, the error is propagated up the call stack.

Adding Context to Errors

Sometimes it's useful to add context to errors to make them more informative:

```rust
use neo::prelude::*;
use std::path::Path;

fn load_wallet_and_get_balance(
    wallet_path: &Path,
    password: &str,
    provider: &Provider,
    token_hash: ScriptHash,
) -> Result<u64, NeoError> {
    // Load the wallet with context
    let wallet = Wallet::load(wallet_path, password)
        .map_err(|e| NeoError::IllegalState(format!("Failed
to load wallet: {}", e)))?;

    // Get the default account with context
    let account = wallet.default_account()
        .map_err(|e| NeoError::IllegalState(format!("Failed
to get default account: {}", e)))?;

    // Create a NEP-17 token instance
    let token = Nep17Contract::new(token_hash,
provider.clone());

    // Get the token balance with context
    let balance = token.balance_of(account.address())
        .map_err(|e| NeoError::IllegalState(format!("Failed
to get token balance: {}", e)))?;

    Ok(balance)
}
```

Using Option to Result Conversion

The NeoRust SDK provides utility functions for converting `Option` to `Result`:

```rust
use neo::prelude::*;

fn get_value_from_option<T>(option: Option<T>, error_message:
&str) -> Result<T, NeoError> {
    option.ok_or_else(||
NeoError::IllegalState(error_message.to_string()))
}

fn example() -> Result<(), NeoError> {
    let optional_value: Option<u64> = Some(42);

    // Convert Option to Result with a custom error message
    let value = get_value_from_option(optional_value, "Value
is None")?;

    // Or use the ok_or_else method directly
    let value = optional_value.ok_or_else(||
NeoError::IllegalState("Value is None".to_string()))?;

    Ok(())
}
```

Converting Between Error Types

The NeoRust SDK provides comprehensive `From` implementations for
converting between different error types:

```rust
// From implementations for domain-specific errors
impl From<BuilderError> for NeoError {
    fn from(err: BuilderError) -> Self {
        // Conversion logic that maps BuilderError variants
to appropriate NeoError variants
    }
}

impl From<CryptoError> for NeoError {
    fn from(err: CryptoError) -> Self {
        // Conversion logic that maps CryptoError variants to
appropriate NeoError variants
    }
}

impl From<WalletError> for NeoError {
    fn from(err: WalletError) -> Self {
        NeoError::WalletError(err)
    }
}

// From implementations for standard library errors
impl From<std::io::Error> for NeoError {
    fn from(err: std::io::Error) -> Self {
        NeoError::IoError(err)
    }
}

impl From<serde_json::Error> for NeoError {
    fn from(err: serde_json::Error) -> Self {
        NeoError::SerializationError(err.to_string())
    }
}

impl From<hex::FromHexError> for NeoError {
    fn from(err: hex::FromHexError) -> Self {
        NeoError::InvalidEncoding(format!("Hex error: {}",
err))
    }
}

impl From<std::num::ParseIntError> for NeoError {
    fn from(err: std::num::ParseIntError) -> Self {
        NeoError::IllegalArgument(format!("Integer parsing
error: {}", err))
    }
}
```

```
        }
    }
```

This allows for easy conversion between error types using the `?` operator. When you use the `?` operator on a function that returns a domain-specific error type, it will be automatically converted to `NeoError` if you're in a function that returns `Result<T, NeoError>`.

Handling RPC Errors

When working with RPC calls, you may need to handle specific error codes:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Try to get a transaction
    let tx_hash =
"0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890
abcdef".parse::<TxHash>()?;

    match provider.get_transaction(&tx_hash).await {
        Ok(tx) => {
            println!("Transaction found: {:?}", tx);
        },
        Err(NeoError::RpcError(RpcError::JsonRpcError { code,
message, .. })) if code == -100 => {
            println!("Transaction not found: {}", message);
        },
        Err(e) => {
            println!("Error: {}", e);
            return Err(e.into());
        }
    }

    Ok(())
}
```

Handling Wallet Errors

When working with wallets, you may need to handle specific wallet errors:

```rust
use neo::prelude::*;
use std::path::Path;

fn open_wallet(wallet_path: &Path, password: &str) ->
Result<Wallet, NeoError> {
    match Wallet::load(wallet_path, password) {
        Ok(wallet) => {
            println!("Wallet loaded successfully");
            Ok(wallet)
        },

Err(NeoError::WalletError(WalletError::InvalidPassword)) => {
            println!("Invalid password");

Err(NeoError::WalletError(WalletError::InvalidPassword))
        },
        Err(NeoError::WalletError(WalletError::FileError(e)))
=> {
            println!("File error: {}", e);

Err(NeoError::WalletError(WalletError::FileError(e)))
        },
        Err(e) => {
            println!("Error: {}", e);
            Err(e)
        }
    }
}
```

Handling Transaction Errors

When sending transactions, you may need to handle specific transaction errors:

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Connect to a Neo N3 TestNet node
    let provider =
Provider::new_http("https://testnet1.neo.coz.io:443");

    // Load your wallet
    let wallet_path = Path::new("my-wallet.json");
    let password = "my-secure-password";
    let wallet = Wallet::load(wallet_path, password)?;

    // Get the account that will send the transaction
    let account = wallet.default_account()?;

    // Create a transaction
    let transaction = TransactionBuilder::new()
        .version(0)
        .nonce(rand::random::<u32>())
        .valid_until_block(provider.get_block_count().await?
+ 100)
        .script(
            ScriptBuilder::new()
                .contract_call(

"d2a4cff31913016155e38e474a2c06d08be276cf".parse::
<ScriptHash>()?,
                    "transfer",
                    &[

ContractParameter::hash160(account.address().script_hash()),

ContractParameter::hash160("NZNos2WqTbu5oCgyfss9kUJgBXJqhuYAa
j".parse::<Address>()?),

ContractParameter::integer(1_00000000), // 1 GAS
                        ContractParameter::any(None),
                    ],
                )
                .to_array()
        )
        .sign(account)?
        .build();
```

```rust
    // Send the transaction
    match provider.send_raw_transaction(&transaction).await {
        Ok(txid) => {
            println!("Transaction sent with ID: {}", txid);
        },

Err(NeoError::TransactionError(TransactionError::Insufficient
Funds)) => {
            println!("Insufficient funds to send the
transaction");
        },
        Err(NeoError::RpcError(RpcError::JsonRpcError { code,
message, .. })) => {
            println!("RPC error: {} (code: {})", message,
code);
        },
        Err(e) => {
            println!("Error: {}", e);
            return Err(e.into());
        }
    }

    Ok(())
}
```

Custom Error Types

You can create custom error types for your application that wrap the NeoRust
SDK errors:

```rust
use neo::prelude::*;
use thiserror::Error;

#[derive(Error, Debug)]
pub enum AppError {
    #[error("Neo SDK error: {0}")]
    NeoError(#[from] NeoError),

    #[error("Configuration error: {0}")]
    ConfigError(String),

    #[error("Database error: {0}")]
    DbError(String),

    #[error("User error: {0}")]
    UserError(String),
}

fn app_function() -> Result<(), AppError> {
    // Use the NeoRust SDK
    let wallet =
Wallet::new("password").map_err(AppError::NeoError)?;

    // Or with the ? operator
    let wallet = Wallet::new("password")?;

    Ok(())
}
```

Error Logging

The NeoRust SDK uses the `tracing` crate for logging errors. You can configure the logging level to see more detailed error information:

```rust
use neo::prelude::*;
use tracing_subscriber::{fmt, EnvFilter};

fn main() {
    // Initialize the logger with custom configuration
    tracing_subscriber::fmt()
        .with_env_filter(EnvFilter::from_default_env()
            .add_directive("neo=debug".parse().unwrap())
            .add_directive("warn".parse().unwrap()))
        .init();

    // Now errors will be logged with more detail
}
```

SGX Error Handling

If you're using the SGX features, there are additional error types for SGX-specific operations:

```rust
pub enum SgxError {
    // Enclave errors
    EnclaveError(sgx_types::sgx_status_t),

    // Attestation errors
    AttestationError,

    // Sealing errors
    SealingError,

    // Other errors
    Other(String),
}
```

Handling SGX errors:

```rust
use neo::prelude::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Path to the enclave shared object
    let enclave_path = "path/to/enclave.so";

    // Initialize the SGX enclave
    match SgxEnclaveManager::new(enclave_path) {
        Ok(enclave_manager) => {
            println!("SGX enclave initialized
successfully!");

            // Use the enclave manager
        },

Err(NeoError::SgxError(SgxError::EnclaveError(status))) => {
            println!("SGX enclave initialization failed with
status: {:?}", status);
        },
        Err(e) => {
            println!("Error: {}", e);
            return Err(e.into());
        }
    }

    Ok(())
}
```

Neo X Error Handling

If you're using the Neo X features, there are additional error types for Neo X-specific operations:

```rust
pub enum NeoXError {
    // EVM errors
    EvmError(String),

    // Bridge errors
    BridgeError(String),

    // Other errors
    Other(String),
}
```

Handling Neo X errors:

```rust
use neo::prelude::*;
use neo::neo_x::evm::*;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Create a Neo X provider
    let provider =
NeoXProvider::new_http("https://rpc.neoX.io");

    // Create a transaction
    let transaction = NeoXTransaction::new()
        .to("0x1234567890123456789012345678901234567890")
        .value(1_000_000_000_000_000_000u128) // 1 ETH in wei
        .gas_price(20_000_000_000u64) // 20 Gwei
        .gas_limit(21_000u64)
        .build();

    // Send the transaction
    match provider.send_transaction(&transaction).await {
        Ok(txid) => {
            println!("Transaction sent with ID: {}", txid);
        },

Err(NeoError::NeoXError(NeoXError::EvmError(message))) => {
            println!("EVM error: {}", message);
        },
        Err(e) => {
            println!("Error: {}", e);
            return Err(e.into());
        }
    }

    Ok(())
}
```

Best Practices

1. **Use the `?` Operator**: Use the `?` operator for concise error propagation.

2. **Match on Specific Errors**: Match on specific error types when you need to handle them differently.
3. **Custom Error Types**: Create custom error types for your application that wrap the NeoRust SDK errors.
4. **Error Logging**: Configure logging to see more detailed error information.
5. **Error Context**: Add context to errors to make them more informative.
6. **Error Recovery**: Implement recovery strategies for recoverable errors.
7. **Error Testing**: Write tests for error conditions to ensure they're handled correctly.
8. **Avoid `unwrap()` and `expect()`**: In production code, avoid using `unwrap()` or `expect()` as they will panic on errors. Instead, use proper error handling with `Result` and the `?` operator.
9. **Convert Domain-Specific Errors**: Use `.into()` to convert domain-specific errors to `NeoError` when needed.
10. **Provide Descriptive Error Messages**: When creating errors, provide descriptive messages that help identify the cause of the error.

Example Code

This section contains examples demonstrating how to use the NeoRust SDK.

Wallet Management

```rust
use neo::prelude::*;
use std::path::Path;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Create a new wallet
    let password = "my-secure-password";
    let wallet = Wallet::new(password)?;

    // Generate a new account
    let account = wallet.create_account()?;
    println!("New account address: {}", account.address());

    // Save the wallet to a file
    wallet.save("my-wallet.json")?;

    Ok(())
}
```

For more examples, see the `examples` directory in the NeoRust repository.