NodeFoundry DePIN Platform - Technical Specification

Document Status: DRAFT
Version: 1.0
Date: August 2025

Overview

NodeFoundry is an aggregation layer for Decentralized Physical Infrastructure Networks (DePINs) that gives users a single interface to connect their wallet and access any DePIN service across any blockchain without friction. NodeFoundry bridges 100+ DePIN and AI networks from any chain to Stellar, creating a single interface and payment layer that democratizes access to affordable, high-performance compute not just for the banked, but anyone with a Stellar wallet.

Core Smart Contracts

1. DePIN Registry Contract

The DePIN Registry contract manages the core infrastructure provider data and serves as the foundation for the entire platform.

Core Methods:

`initialize`

```
initialize(admin: Address)
```

Initializes the contract with an admin address who has permission to manage DePIN providers.

`add_depin`

```
add_depin(
    invoker: Address,
    name: String,
    description: String,
    uptime: i32,
    reliability: i32,
    cost: i32
) -> BytesN<32>
```

Parameters: | Name | Type | Description | |——|——|————-| | invoker | Address | Admin address calling the function | | name | String | DePIN provider name | | description | String | Detailed description of services | | uptime | i32 | Uptime percentage (0-100) | | reliability | i32 | Reliability score (0-100) | | cost | i32 | Cost per hour in smallest token unit |

The `add_depin` method stores the DePIN provider data in contract storage using a unique `BytesN<32>` ID as the primary key. The ID is generated using an incremental counter encoded in the first 4 bytes.

We also store all DePIN IDs in a separate collection to enable easy listing of all providers on the marketplace interface.

`update_depin`

Updates existing DePIN provider information (admin only).

`get_depin`

```
get_depin(depin_id: BytesN<32>) -> Option<DePIN>
```

Retrieves DePIN provider data using the ID passed as parameter.

`list_depins`

```
list_depins() -> Vec<BytesN<32>>
```

Returns all DePIN provider IDs. The frontend can then call `get_depin` for each ID to retrieve full provider data.

`set_depin_status`

```
set_depin_status(invoker: Address, depin_id: BytesN<32>, status: bool)
```

Enables or disables DePIN providers (admin only).

---

2. Reputation Contract

The reputation contract manages user reviews and ratings for DePIN providers, creating a trust layer for the marketplace.

Core Methods:

`initialize`

```
initialize(admin: Address, depin_registry_address: Address)
```

Links the reputation contract to the DePIN registry for validation.

`rate_and_review_depin`

```
rate_and_review_depin(
    invoker: Address,
    depin_id: BytesN<32>,
    rating: i32,
    review: String
)
```

Parameters: | Name | Type | Description | |——|——|————-| | invoker | Address | User submitting the review | | depin_id | BytesN<32> | ID of the DePIN provider | | rating | i32 | Rating score (1-5) | | review | String | Text review from user |

The method stores review data as a tuple `(Address, i32, String)` mapped to the DePIN ID. Users can update their existing reviews.

`get_reviews`

```
get_reviews(depin_id: BytesN<32>) -> Vec<(Address, i32, String)>
```

Returns all reviews for a specific DePIN provider.

`get_average_rating`

```
get_average_rating(depin_id: BytesN<32>) -> i32
```

Calculates and returns the average rating for a DePIN provider.

---

3. User Profile Contract

The user profile contract manages user accounts, wallet functionality, and subscription systems.

Core Methods:

`create_user_profile`

```
create_user_profile(
    user_address: Address,
    username: String,
    email: String,
    referral_code: Option<String>
)
```

Parameters: | Name | Type | Description | |——|——|————-| | user_address | Address | User's Stellar address | | username | String | Chosen username | | email | String | User email address | | referral_code | Option | Optional referral code |

Creates a comprehensive user profile with wallet functionality and referral tracking.

`deposit_funds`

```
deposit_funds(
    user_address: Address,
    token_address: Address,
    amount: i128
)
```

Allows users to deposit supported tokens (USDC by default) into their platform wallet.

`withdraw_funds`

```
withdraw_funds(
    user_address: Address,
    token_address: Address,
    amount: i128
)
```

Enables users to withdraw funds from their platform wallet.

`subscribe_to_plan`

```
subscribe_to_plan(
    user_address: Address,
    plan: SubscriptionPlan
)
```

Subscription Plans: - **Basic**: Free tier with limited access - **Premium**: 10 USDC/month with priority access - **Enterprise**: 50 USDC/month with full features

Subscription costs are automatically deducted from user balances. Users earn loyalty points based on spending patterns.

`has_sufficient_balance`

```
has_sufficient_balance(
    user_address: Address,
    amount: i128
) -> bool
```

Utility function used by other contracts to validate user balances before transactions.

---

4. Order Management Contract

The order contract handles service requests with an escrow mechanism to ensure secure transactions.

Core Methods:

`create_order`

```
create_order(
    user: Address,
    depin_id: BytesN<32>,
    service_type: String,
    duration_hours: u64,
    price_per_hour: i128,
    deployment_chain: String,
    service_params: String
) -> BytesN<32>
```

Parameters: | Name | Type | Description | |——|——|————-| | user | Address | User placing the order | | depin_id | BytesN<32> | Target DePIN provider | | service_type | String | Type of service (compute, storage, etc.) | | duration_hours | u64 | Service duration | | price_per_hour | i128 | Hourly rate | | deployment_chain |

String | Target blockchain for deployment | | service_params | String | JSON-encoded service parameters |

The method validates user balance, creates an escrow by deducting funds, and generates a unique order ID.

`complete_order`

```
complete_order(provider: Address, order_id: BytesN<32>)
```

Releases escrowed funds to the DePIN provider upon successful service completion.

`cancel_order`

```
cancel_order(user: Address, order_id: BytesN<32>)
```

Cancels pending orders and refunds escrowed funds to users.

`update_order_status`

```
update_order_status(
    provider: Address,
    order_id: BytesN<32>,
    status: OrderStatus
)
```

Allows providers to update order status (Pending, InProgress, Completed, Failed).

Staking Contract

A dedicated staking contract will be developed to handle:

Core Methods:

`stake_tokens`

```
stake_tokens(
    user: Address,
    provider_id: BytesN<32>,
    amount: i128,
    lock_period: u64
)
```

Allows users to stake tokens with DePIN providers for rewards.

`unstake_tokens`

```
unstake_tokens(user: Address, stake_id: BytesN<32>)
```

Enables users to withdraw staked tokens after lock period.

`distribute_rewards`

```
distribute_rewards(provider_id: BytesN<32>)
```

Distributes rewards to stakers based on provider performance and reputation.

Leaderboard Contract

Core Methods:

`update_user_score`

```
update_user_score(user: Address, activity_type: String, points: i32)
```

Updates user scores based on platform activity.

`get_leaderboard`

```
get_leaderboard(limit: u32) -> Vec<(Address, i32)>
```

Returns top users by score.

Cross-Contract Integration

The contracts are designed to work together through cross-contract calls:

1. **Order Contract** validates user existence and balance through **User Profile Contract**
2. **Reputation Contract** validates DePIN existence through **Registry Contract**
3. **Staking Contract** will integrate reputation data for reward calculations

---

Data Storage Patterns

All contracts use Soroban persistent storage with enum-based keys:

```
#[contracttype]
pub enum DataKey {
    Admin,
    DepinMap,
    Counter,
    // Additional keys as needed
}
```

IDs are generated using incremental counters encoded in `BytesN<32>` format for consistency across contracts.

Security Considerations

1. **Admin Controls**: All administrative functions require proper authorization
2. **Input Validation**: Strict parameter validation on all contract methods

3. **Escrow Protection**: Funds are secured until service completion
4. **Cross-Contract Validation**: Contracts validate data with each other
5. **Error Handling**: Comprehensive error messages for debugging

---