The social platform that realizes blockchain authentication even in environments without the internet

table of contents

1. Certificate issuing tool [1]

1.1 Function Description

This system is a web-based tool that utilizes the Cardano blockchain to issue and manage verifiable certificates for disaster relief volunteers. The certificates include detailed information such as volunteers' skills, roles, and authorized regions, enabling reliable and transparent qualification verification in disaster response situations. The main functions of the system are listed below.

1.1.1 Certificate Registration Function [1-a]

1.1.1.1 Creation and Issuance of Qualification Certificates

- Users enter volunteer qualification information (e.g., name, role, skills, available activity regions, etc.) through the frontend interface to create certificate data structured in JSON format.
- The certificates comply with the "Verifiable Credential" format, ensuring a reliable structure. This certificate data is recorded on the blockchain as verification information, allowing others to reference and verify it in the future.

1.1.1.2 Transaction Generation and Signing

- A transaction is generated using the CardanoWasm (Cardano Web Assembly) library, and credential data is set as auxiliary data. The credential data is serialized to be recorded on the blockchain and signed with the user's private key.
- The signature prevents tampering with the transaction and ensures the reliability of the credential. Through this process, the user's credential data is uniquely included in the transaction.

1.1.1.3 Transaction Submission and Recording on the Blockchain

- The signed transaction is submitted via the backend server and recorded on the Cardano blockchain. The backend server manages the transaction submission using the Blockfrost API, and upon successful recording, a transaction ID is returned.
- Using this transaction ID, it is possible to verify that the submitted credential data has been recorded in the block.

1.1.2 Function to Retrieve Transaction Information [1-b]

1.1.2.1 Retrieving Block Information and Verification via Merkle Tree

- After a transaction is included in a block, a Merkle tree is constructed based on the block information and transaction ID to verify the credential data.
- The hash of each transaction is stored in the Merkle tree, ensuring the validity and integrity of the transaction. This procedure proves that a specific credential is securely recorded in the correct block on the Cardano blockchain.

1.1.2.2 Downloading Credential Data

- Ultimately, the credential data, including the Merkle tree path and block information, can be downloaded in JSON format. This allows users to refer to and verify the credential data later, while also enabling third parties to verify the integrity of the credential data even in an offline environment.

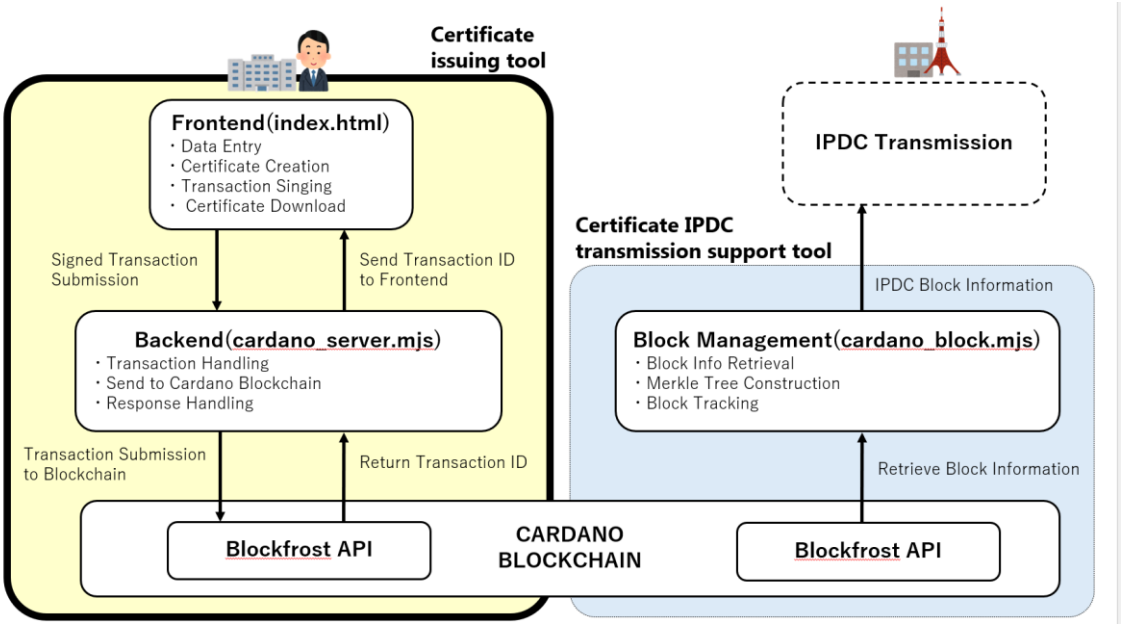
Through this system, volunteer credentials can be quickly and reliably verified in disaster relief situations. Additionally, blockchain technology ensures protection against credential forgery, enhancing the credibility of volunteer qualifications. Disaster response agencies can confirm volunteer credentials with confidence, even without scanning cards or online access, thereby improving the efficiency and safety of disaster response activities.

1.2 System Architecture

The certificate issuance tool integrates with the frontend, backend, and external APIs (Blockfrost API) to comprehensively handle the process from user input to certificate generation, signing, transaction submission, and retrieval of verification data. This architecture ensures the generation of credential data and guarantees its reliability on the blockchain.

1.2.1 System Architecture Diagram

This system consists of a blockchain information update and management module (cardano_block.mjs) and operates in coordination with the Cardano blockchain and the Blockfrost API. It periodically retrieves IPDC block information from the Cardano blockchain and provides the data to the broadcasting station for IPDC transmission.



1.2.2 System Components

1.2.2.1 Frontend

- **Role**

The frontend is responsible for inputting credential data, generating and signing transactions, and communicating with the backend.

- **Key Components**

1. **Data Input Form**

   A text area where users enter credential information in JSON format.

2. **Transaction Generation Function**

   Uses the CardanoWasm（Cardano Web Assembly）library to generate and sign a transaction based on user-inputted data, creating a signed transaction.

3. **Backend Communication Function**

   Sends the signed transaction to the backend's /submit endpoint, requesting transaction submission via the Blockfrost API.

4. **Download Function**

   Once the transaction is successfully recorded, users can download the credential data in JSON format.

1.2.2.2 Backend (cardano_server.mjs)

- **Role**

The backend is responsible for recording signed transactions received from the frontend onto the Cardano blockchain.

- **Key Components**

1. **Data Input Form**

   The backend is responsible for recording signed transactions received from the frontend onto the Cardano blockchain.

2. **Submit Endpoint**

   Receives POST requests from the frontend and passes the signed transaction data to the sendTransaction function.

3. **Transaction Submission Function**

   The sendTransaction function sends the transaction via the Blockfrost API, retrieves the transaction ID, and returns it to the frontend.

4. **Error Handling**

   If the transaction submission fails, an error response containing a message is sent to the frontend to prompt appropriate action.

1.2.2.3 Blockfrost API

- **Role**

The Blockfrost API receives transaction submissions from the backend and records them on the Cardano blockchain. It also provides functionality for retrieving and verifying block information.

- **Key Features**

1. **Transaction Submission (/tx/submit Endpoint)**

   Accepts signed transactions and records them on the blockchain. Upon success, it returns a transaction ID, which can be used for verification based on block information.

2. **Block Information Retrieval**

   Retrieves block information, hashes, and Merkle tree data based on the transaction ID, enabling the verification of credential data integrity.

1.2.3 System Data Flow

1. **Credential Data Input**

   The user enters credential information in the frontend.

2. **Transaction Generation and Signing**

   The frontend generates and signs a transaction.

3. **Transaction Submission**

   The signed transaction data is sent to the backend.

4. **Blockchain Recording via Blockfrost API**

   The backend submits the transaction via the Blockfrost API and retrieves the transaction ID.

5. **Block Information Retrieval and Verification**

   After the transaction is recorded, the user can download verification data.

1.2.4 System Architecture Features

1. Separation of Roles

The frontend handles user input and transaction generation, the backend manages transaction submission and error handling, and the Blockfrost API is responsible for blockchain recording. This modular approach ensures a consistent data flow with clear role distribution.

2. Security and Reliability

Credential data is signed and recorded on the blockchain, ensuring tamper resistance. Additionally, integration with the Blockfrost API enables easy retrieval of blockchain information and verification of transaction integrity.

3. Downloadable Data in a Verifiable Format

Data, including the transaction ID and block information, can be downloaded in JSON format, allowing for reliable data reference in disaster relief situations.

With this architecture, the certificate issuance tool ensures a seamless process from credential data input to transaction recording and verification, maintaining data security and reliability throughout.

1.3 Module Design

1.3.1 Frontend

The frontend provides an interface for creating and managing disaster relief volunteer certificates. It enables certificate data input, transaction generation and signing, submission to the backend server, retrieval of block information, verification using a Merkle tree, and certificate data download. The main functionalities and their roles are as follows:

1.3.1.1 Transaction Metadata Input

- **Elements**

  <textarea id=”metadataInput”>

- **Description**

  A text area for users to input volunteer credential information in JSON format. The JSON follows the ”Verifiable Credential” format, containing details such as the volunteer’s role, skills, and authorized regions.

- **Behavior**

  The inputted credential data is encoded as **Auxiliary Data** in the transaction. This data is later included in the transaction and sent to the blockchain for permanent storage.

1.3.1.2 Transaction Submission Button

- **Element**

  <button id="submitBtn">Submit Transaction</button>

- **Description**

  A button for submitting transactions. When clicked, it triggers transaction generation, signing, and submission to the backend.

- **Behavior**

  Clicking the button initiates the **transaction creation, signing, and submission process** with the following steps:

  1. **Address Generation**
     Uses a hardcoded private key to generate a public key and address.
  2. **Retrieve Maximum UTXO**
     Fetches Unspent Transaction Outputs (UTXOs) associated with the user's address and selects the largest UTXO.
  3. **Transaction Generation**
     Calculates the transfer amount and transaction fee based on the maximum UTXO. Constructs a transaction using the generated address.
  4. **Metadata Encoding**
     Encodes the inputted metadata from the frontend and adds it to the transaction as Auxiliary Data.
  5. **Transaction Signing**
     Signs the generated transaction using the private key. Produces a signed transaction in **Hex format.**
  6. **Submission to Backend**
     Sends the signed transaction data as a JSON object to the backend server. Receives a transaction ID from the backend upon successful submission.

1.3.1.3 Retrieving Block Information and Generating Merkle Tree

- **Functions**

  fetchBlockInfo()
  fetchBlockTransactions()
  buildMerkleTree()
  findPath()
  verifyMerklePath()

- **Description**

  Retrieves block information from the Blockfrost API based on the transaction ID.

  Generates a Merkle tree from the transaction hash and constructs a Merkle path for verification.

  Uses the verifyMerklePath function to check whether a specific transaction is legitimately recorded in the Merkle tree.

- **Purpose**

  Ensures that the transaction is correctly recorded on the blockchain. Guarantees the integrity and authenticity of the certificate data.

1.3.1.4 Certificate Data Download

- **Element**

  <button id=”vc_download” disabled>Download Certificate</button>

- **Description**

  A button for downloading the credential certificate data in JSON format. It remains disabled until the transaction is recorded and verified on the blockchain.

- **Behavior**

  Once the transaction and block information are retrieved, the certificate data (including certificate details, transaction Merkle path, etc.) is generated. The vc_download button becomes enabled. When the user clicks the button, the certificate data is downloaded as a JSON file (vc.json).

1.3.1.5 Main Data Flow

- **Data Flow ①**

  Certificate Data Input
  → **Transaction Generation & Signing**
  → **Submission**

  The user inputs credential data, and a signed transaction is generated. The signed transaction is sent to the backend, where it is recorded on the blockchain.

- **Data Flow ②**

  Transaction ID Reception
  → **Block Information Retrieval & Merkle Tree Generation**
  → **Certificate Data Download**

  Based on the received transaction ID, the system retrieves block information and generates the Merkle tree path. The verified certificate data is then made available to the user for download.

This frontend design enables users to seamlessly complete the entire process—from certificate data entry to blockchain recording, verification, and certificate download. Disaster relief organizations can easily and securely verify volunteer credentials, ensuring the trustworthiness of volunteer activities.

1.3.2 Backend

The backend server is built using the Express framework. It is responsible for receiving signed transactions from the frontend, submitting them to the blockchain, and managing transaction results. Additionally, it serves as an intermediary that interacts with the Blockfrost API to ensure proper transaction recording.

1.3.2.1 Server Configuration and Dependencies

- **Dependencies**
  - `express`: Web server framework.
  - `body-parser`: Parses request bodies.
  - `axios`: Sends HTTP requests to external APIs (e.g., Blockfrost API).
  - `dotenv`: Loads environment variables (e.g., API keys).
  - `log4js`: Manages logging for error messages and transaction processing.
- **Configuration**
  - Loads the Blockfrost API key from the .env file. Initializes logging when the server starts.
  - Supports logging configurations to record error messages and transaction processing statuses in both files and console output.

1.3.2.2 Logging Configuration

- **Log File**

  File Path: logs/cardano_server.log

- **Configuration**

  Uses log4js to set up both file logging and console logging. Specifies maximum log file size and number of backup files to maintain.

- **Purpose**

  Enhances tracking of transaction processing and error handling. Facilitates debugging by making it easier to identify transaction flows and pinpoint issues.

1.3.2.3 Endpoint

/submit (POST)

- **Description**

  Receives a signed transaction from the frontend and sends it to the Cardano network.

- **Processing Flow**

  1. **Request Handling**

     Extract the **signed transaction data (Hex format)** from the payload field.

  2. **Transaction Submission**

     Call the sendTransaction function to send the transaction to the Cardano blockchain using the Blockfrost API.

  3. **Response Return**

     Return the Blockfrost API response to the frontend:

     On success: Returns the transaction ID.

     On failure: Returns an error message.

/submit (GET)

- **Description**

  Receives the payload query parameter and executes the transaction submission process (for debugging and testing purposes).

- **Processing Flow**

  1. Retrieve the payload parameter (signed transaction) from the query parameters and call the sendTransaction function.
  2. Return the processing result as an HTTP response.

1.3.2.4 sendTransaction Function

- **Parameters**

  signedTxHex: The signed transaction data in Hex format.

- **Processing**

  Encode the signed transaction data in CBOR format. Send a POST request to the Blockfrost API's /tx/submit endpoint.

  Upon success, retrieve the transaction ID and return the response data for error handling.

- **Error Handling**

  If the transaction submission fails, return the error message and response code.

  Log the error details in the console and server logs. Send an appropriate error response to the frontend.

1.3.2.5 Error Handling

- **Configuration**

  If a network error or authentication error occurs, it is caught within the sendTransaction function. An HTTP 500 status code along with an error message is sent to the frontend.

- **Purpose**

  Ensures that accurate error messages are returned to the frontend. Allows users to be notified of error details, improving debugging and user experience.

1.3.2.6 CORS Configuration

- **Purpose**

  Allows cross-origin requests (CORS) from the frontend. Enables requests between different origins to ensure proper communication between the frontend and backend.

1.3.2.7 Server Listening

- **Configuration**

  The server **listens for requests on port 1337** using:
  app.listen(1337);
  This ensures the server is **actively running** and ready to handle incoming requests.

1.3.2.8 Backend Role and Data Flow

The backend server acts as an intermediary that processes transaction submission requests from the frontend and sends credential certificate data to the Cardano blockchain.

Data Flow

1. Receives a transaction submission request from the frontend.
2. Sends the signed transaction to the Cardano blockchain via the Blockfrost API.
3. On success, returns the transaction ID to the frontend.
4. The transaction ID is then used for block verification and Merkle tree validation.

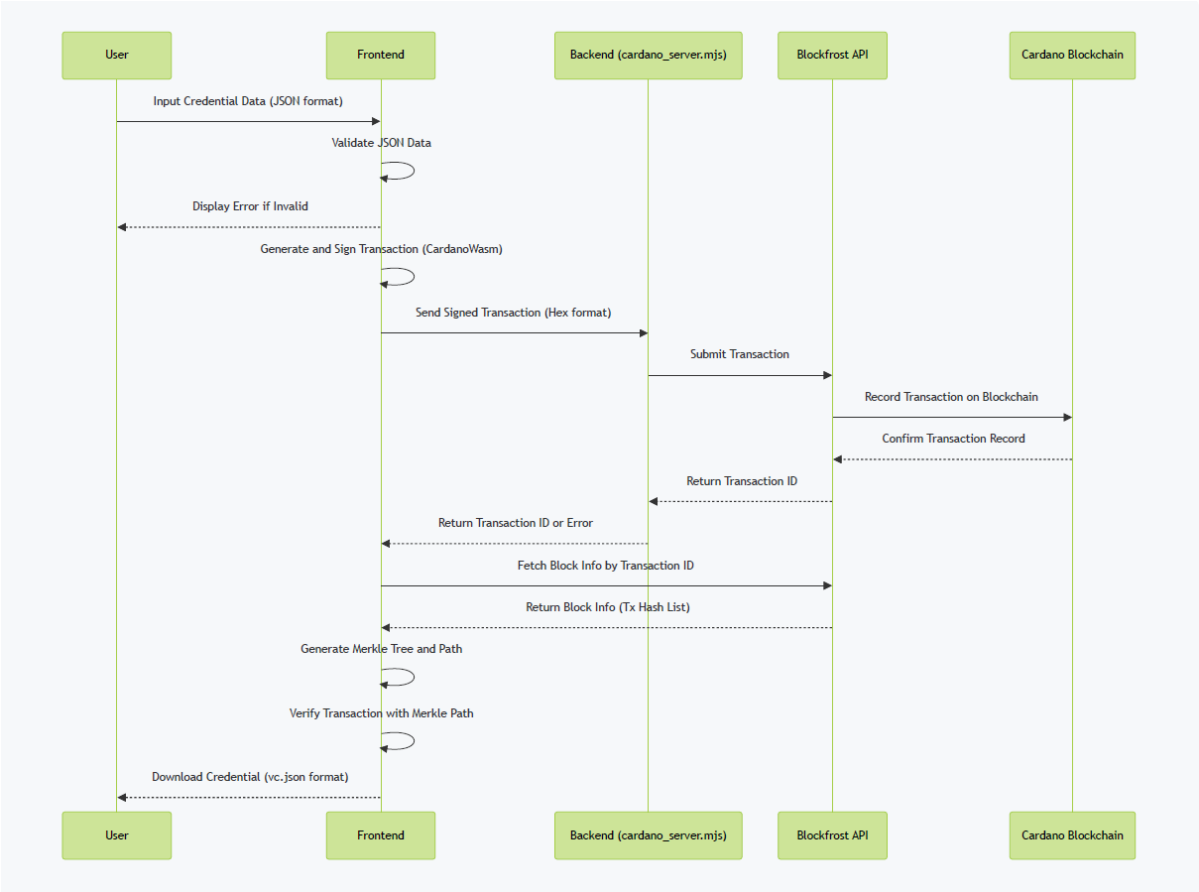
By implementing robust error handling, the backend enhances system reliability and improves user experience.

1.4 Data Flow and Algorithm Design

1.4.1 Data Flow and Algorithm

The certificate issuance tool provides an end-to-end process that seamlessly handles: User input、Generation of credential data、Transaction signing and submission、Recording on the blockchain、Retrieval and download of verification data. Below is the detailed data flow and algorithm for each step.

Data Flow Diagram



1.4.1.1 Credential Data Input

- **Processing Details**

  The user inputs credential data (e.g., name, role, skills, authorized regions) in JSON format. The user submits the data by clicking the submit button on the frontend interface.

- **Data Validation**

  The frontend verifies whether the input follows the correct JSON format. If the format is invalid, an error message is displayed. If the format is valid, the system proceeds to transaction generation.

1.4.1.2 Transaction Generation and Signing

- **Algorithm**

  Uses CardanoWasm（Cardano Web Assembly）library to encode credential data as Auxiliary Data and generate a transaction.

- **Detailed Steps**

  Address Generation

  Derives the public key from the user's private key. Retrieves the corresponding Cardano address.

  Fetching UTXO

  Retrieve the UTXOs (Unspent Transaction Outputs) associated with the user's address and select the largest UTXO for use.

  Transaction Generation

  Calculates transfer amount and transaction fees based on the selected UTXO. Constructs the transaction and adds the credential data as Auxiliary Data.

  Signing Process

  Signs the generated transaction using the user's private key. Produces a signed transaction in Hex format.

1.4.1.3 Transaction Submission

- **Backend Processing**

  The frontend sends a signed transaction to the /submit endpoint of cardano_server.mjs. The sendTransaction function forwards the transaction to the Blockfrost API, attempting to record it on the Cardano blockchain.

- **Data Validation**

  The frontend ensures that the transaction data is in correct JSON format before submission. If the format is invalid, an error message is displayed. If valid, the system proceeds with transaction submission.

- **On Success**

  The Blockfrost API returns a transaction ID. The backend sends this transaction ID back to the frontend for further processing.

- **On Failure (Error Handling)**

  If transaction submission fails, an error message is sent to the frontend. The user is prompted to retry the submission.

1.4.1.4 Retrieving Block Information and Verifying via Merkle Tree

- **Processing Details**

  Once a transaction is included in a block, the block information is retrieved from Blockfrost API using the transaction ID.

- **Algorithm: Merkle Tree Generation**

  1. Retrieve the list of transaction hashes in the target block from Blockfrost API.
  2. Pair and hash each transaction hash using SHA3-256, forming an upper-level hash.
  3. Repeat the process recursively until reaching a single root hash, constructing the Merkle tree.

- **Algorithm: Path Generation and Verification**

  1. **Generate the Merkle Path**

     The findPath function identifies the path from the transaction hash to the Merkle root.

  2. **Verify the Merkle Path**

     The verifyMerklePath function checks whether the specified transaction exists in the block using the Merkle tree.

1.4.1.5 Certificate Data Download

- **Processing Details**

  The credential certificate data is structured, including the transaction ID, block information, and Merkle tree path. The data is provided to the user in JSON format.

- **Download Format**

  The data can be downloaded as a JSON file named vc.json. The file can be used offline to verify credential information.

- **Purpose**

  Since the integrity of the certificate data is ensured, it can be used for credential verification in disaster relief scenarios and other applications.

1.4.2 Algorithm Overview

- **Transaction Generation and Signing**

  Uses the CardanoWasm library to encode user-input credential data and include it as part of the transaction.

  Calculates the transfer amount and transaction fee based on the largest UTXO. Signs the generated transaction with the user's private key, producing a signed transaction data.

- **Merkle Tree Generation and Path Verification**

  Retrieves the list of transaction ID hashes from the Blockfrost API to construct a Merkle tree.

  Generates the Merkle path from the specific transaction hash to the root hash. Uses this path to verify the data integrity, ensuring the transaction is correctly recorded on the blockchain.

1.4.3 Summary

- **Frontend**

  Handles credential data input and validation. Generates and signs transactions. Sends signed transactions to the backend.

- **Backend**

  Submits signed transactions to the blockchain. Retrieves the transaction ID. Manages error handling.

- **Blockfrost API**

  Records transactions on the blockchain. Retrieves block information. Performs data verification using the Merkle tree.

1.5 Data Design

The following details the data design of the certificate issuance tool. This data design specifies the role, format, and storage method of each data item used throughout the system, ensuring data consistency and reliability across all components.

Since the certificate issuance tool records user-inputted credential data on the blockchain and manages it in a verifiable format, various data items are handled across the frontend, backend, Blockfrost API, and blockchain. The following sections describe the role and design of each data item.

1.5.1 Frontend Data Items

1.5.1.1 **metadataInput**

- **Description**

  This is the input field where the user enters credential certificate data. The user provides information such as:

  Volunteer role、Skills、Authorized regions、Certification details

- **Format**

  JSON format

- **Data Example**

```
{
   "name": "John Doe",
   "role": "Heavy Machinery Operator",
   "skills": ["Excavator Operation", "Bulldozer Operation"],
   "authorizedRegions": ["Prefecture A, City B"]
}
```

1.5.1.2 signedTxHex

- **Description**

  A Hex-encoded signed transaction that includes credential certificate data. This signed transaction is sent from the frontend to the backend for submission to the Cardano blockchain. Once received by the backend, it is ready for blockchain recording.

- **Format**

  Hex-encoded string

- **Example Data**

  ```
  "abcdef1234567890abcdef1234567890"
  ```

1.5.1.3 transactionId

- **Description**

  A unique identifier returned when a transaction is successfully recorded on the blockchain. Used for credential data verification by retrieving block information and Merkle tree validation.

- **Format**

  String

- **Example Data**

  ```
  "abc123def456"
  ```

1.5.1.4 downloadableCertificate

- **Description**

  A JSON-formatted file that allows users to download credential certificate data recorded on the blockchain. Includes the transaction ID, Merkle tree information, and certificate details. Can be used for offline verification of credential authenticity.

- **Format**

  JSON format

- **Example Data**

```
{
   "transactionId": "abc123def456",
   "credentialSubject": { "name": "John Doe", "role": "Operator" },
   "merklePath": [ /* Merkleパス情報 */ ]
}
```

1.5.2 Backend Data Items

1.5.2.1 payload

- **Description**

  The signed transaction data sent from the frontend as a POST request. This data is passed to the Blockfrost API, where it is recorded on the blockchain.

- **Format**

  JSON format

- **Example Data**

```
{
   "payload": "abcdef1234567890abcdef1234567890"
}
```

1.5.2.2 Response Data from Blockfrost API

- **Description**

  The response data returned from the Blockfrost API after submitting a transaction.

  - On success: Includes the transaction ID.
  - On failure: Contains an error message.

- **Format**

  JSON format

- **Example Data(Successful Transaction Response)**

```
{
   "transactionId": "abc123def456"
}
```

- **Example Data(Failed Transaction Response)**

```
{
   "error": "Transaction submission failed"
}
```

1.5.3 Blockfrost API Related Data Items

1.5.3.1 transactionId

- **Description**

  A unique identifier returned by the Blockfrost API when a transaction is successfully recorded on the Cardano blockchain. Later used to retrieve block information and verification data.

- **Format**

  String

- **Example Data**

  `"abc123def456"`

1.5.3.2 blockHeight

- **Description**

  The block height where the transaction was recorded. Required for transaction verification using a Merkle tree.

- **Format**

  Numeric (Integer)

- **Example Data**

  `123456`

1.5.3.3 transactionHashes

- **Description**

  A list of all transaction ID hashes included in a specific block. Used for Merkle tree construction to verify transaction integrity.

- **Format**

  Array of strings

- **Example Data**

```
["hash1", "hash2", "hash3", "..."]
```

1.5.3.4 merklePath

- **Description**

  The Merkle path corresponding to the transaction ID of the credential certificate data. Used to verify data integrity by reconstructing the Merkle tree and confirming the transaction's validity in the block.

- **Format**

  Array of strings

- Example Data

```
[
   {"position": "left", "hash": "hash1"},
   {"position": "right", "hash": "hash2"}
]
```

1.5.4 Certificate Data Items

The credential certificate data structure follows the Verifiable Credential (VC) format, ensuring compliance with decentralized identity standards. It includes the following fields:

1.5.4.1 @context

- **Description**

  A schema URL that defines the data structure of the credential certificate. Helps ensure compatibility with Verifiable Credential standards.

- **Format**

  URL format

- **Example Data**

  ```
  "https://www.w3.org/2018/credentials/v1"
  ```

1.5.4.2 id

- **Description**

  A unique identifier for the credential certificate. Typically formatted as a URN (Uniform Resource Name) or UUID (Universally Unique Identifier).

- **Format**

  String (URN or UUID format)

- **Example Data**

  ```
  "urn:uuid:12345-67890"
  ```

1.5.4.3 type

- **Description**

  Specifies the type of certificate. Typically includes ”VerifiableCredential” along with a custom certificate type.

- **Format**

  Array of strings

- **Example Data**

```
["VerifiableCredential", "DisasterReliefVolunteerCredential"]
```

1.5.4.4 issuer

- **Description**

  Contains information about the entity that issued the credential certificate. This can be an organization, government entity, or decentralized identity (DID).

- **Format**

  Object

- **Example Data**

```
{
   "id": "https://disaster-response.example.org",
   "name": "Disaster Response Headquarters"
}
```

1.5.4.5 issuanceDate

- **Description**

  The date when the certificate was issued. Follows the ISO 8601 date format for standardization.

- **Format**

  ISO 8601 format (YYYY-MM-DDTHH:MM:SSZ)

- **Example Data**

  `"2024-11-03T00:00:00Z"`

1.5.4.6 credentialSubject

- **Description**

  Contains information about the subject (the individual or entity receiving the certificate). Includes name, role, skills, and authorized regions for validation.

- **Format**

  Object

- **Example Data**

```
{
   "id": "did:example:volunteer-001",
   "name": "John Doe",
   "role": "Operator",
   "skills": ["Excavator Operation"],
   "authorizedRegions": ["Prefecture A, City B"]
}
```

1.5.5 Summary

This data design ensures consistent data management across the system, maintaining the reliability and verifiability of credential certificate data.

- **Frontend**

  Receives user-inputted credential information in JSON format. Generates a signed transaction that includes the credential data.

- **Backend**

  Submits the signed transaction to the Cardano blockchain via Blockfrost API. Manages response data, including the transaction ID and error handling.

- Blockfrost API

  Provides block information and Merkle tree data based on the transaction ID. Supports credential certificate verification.

- **Credential Data**

  Follows a verifiable certificate format. Stored on the blockchain, ensuring it remains tamper-proof and secure.

2. Certificate IPDC transmission support tool [2]

2.1 Function Description

This system is a web-based tool that leverages the Cardano blockchain to issue and manage verifiable certificates for disaster relief volunteers.

Each **certificate** includes detailed information such as:
・Volunteer skills
・Volunteer Roles
・Volunteer Authorized regions

By using this system, disaster response organizations can conduct reliable and transparent credential verification, ensuring trust and efficiency in emergency operations.

Key features of the system include:

2.1.1 IPDC Transmission Data Construction Function [2-a]

2.1.1.1 Periodic Block Information Retrieval

- The tool retrieves the latest block information from the Cardano blockchain every 15 seconds. It uses the Blockfrost API to fetch block data, including:
  ・Block height
  ・Block hash
  ・Previous block information
- A file lock mechanism ensures that if a specified file already exists, the data retrieval process is skipped. This prevents conflicts when running concurrent processes.

2.1.1.2 Block Integrity Check and Anomaly Detection

- The system verifies whether the **latest block information** matches the **previously retrieved block** and ensures the **block hash is correct**. If an **anomaly is detected**, a **reconciliation process** is triggered, utilizing historical block data to verify and maintain **data consistency**.
- In case of **discrepancies**, an **anomaly message** is logged to ensure **data reliability and trustworthiness**.

2.1.1.3 Generation and Storage of IPDC Hash Information

- Each block is assigned IPDC-compliant hash information, allowing the next block to reference the previous IPDC hash in accordance with the IPDC format. The sha3_256 algorithm is used to generate the IPDC hash for each block.
- The new IPDC hash is calculated by combining:
  - The previous IPDC hash
  - The current block's transaction information
- The computed IPDC hash information is stored in a file and can be used as proof data for each block.

2.1.1.4 Merkle Root Calculation for Transactions

- Retrieves the list of transaction IDs from the latest block and generates a Merkle tree to verify the integrity of block data.
- The Merkle root hash is added to each block and is also used in IPDC hash calculations.
- If no transactions exist in a block, an empty Merkle root hash is set to ensure data integrity.

2.1.1.5 Certificate File Creation

- Block information and the computed IPDC hash are stored in a JSON file, allowing for later reference and verification.
- A certificate file is generated for each block height, ensuring data traceability.
- The latest file name is appended to trigger.txt, which serves as a trigger file for integration with other systems.

2.1.1.6 Log Management and Error Handling

- Uses the **Log4js library** to log key steps in the process, including:
  - **Tool startup**
  - **Data retrieval**
  - **Anomaly detection**
  - **IPDC hash generation**
  - **File creation**
- These logs help with **troubleshooting and debugging**.
- When an **error occurs**, a **detailed error message** is logged to facilitate **problem detection and resolution**.

2.1.1.7 Automatic Deletion of Old Files

- **Automatically deletes old files** that have exceeded a **specified retention period**, ensuring **efficient storage management**.
- By **default**, file deletion is executed **once per day**, keeping the **storage directory clean and optimized**.

2.1.1.8 Use Cases

- **Real-time Blockchain Data Monitoring**
  Periodically retrieves the latest Cardano blockchain data and generates certificates in IPDC hash format, ensuring that each block's data is linked. Instant anomaly detection helps maintain blockchain data integrity by identifying inconsistencies in real time.
- **Data Proof via IPDC Integration**
  Manages blockchain data in IPDC format, ensuring trustworthiness when referenced by other systems. The generated certificate files provide tamper-proof records, enabling decentralized data verification.

Thus, the Certificate IPDC Transmission Support Tool is designed to periodically retrieve and verify data on the Cardano blockchain while ensuring high reliability and data integrity.

Through IPDC-format hash management, file storage, and anomaly detection, this tool enables secure and consistent block data management.

2.2 System Architecture

The Certificate IPDC Transmission Support Tool is a system that periodically retrieves the latest block information from the Cardano blockchain and assigns an IPDC-format hash to each block to ensure data linkage consistency.

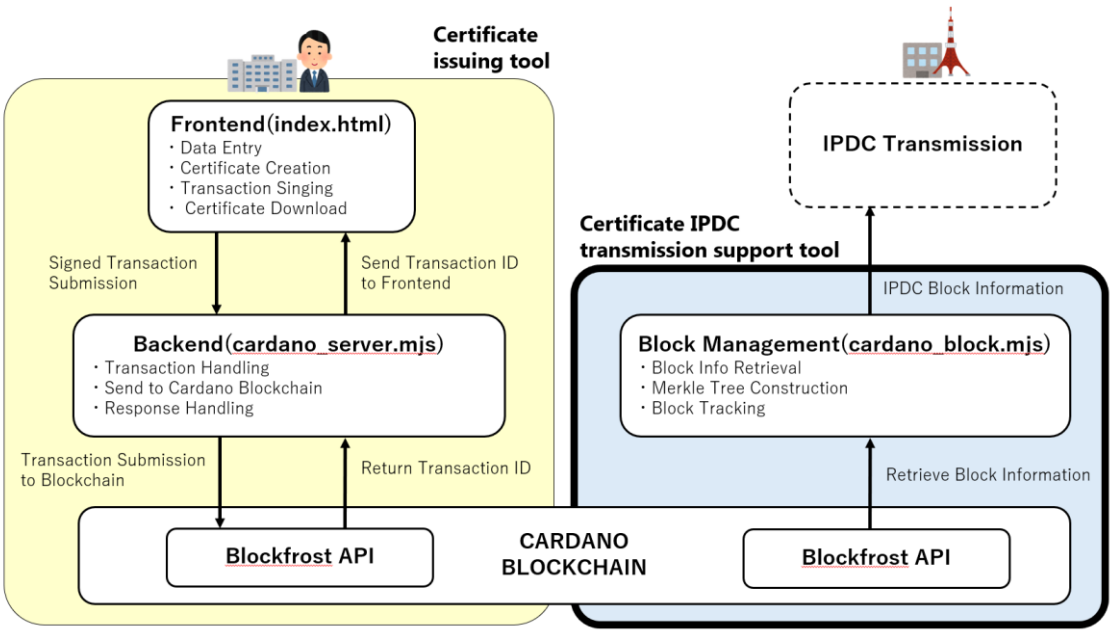
This tool:

・ Manages and stores retrieved block information as IPDC hashes.

・ Generates certificate files in a format that allows integration with other systems.

2.2.1 System Architecture Diagram

This system consists of a blockchain information update and management module (cardano_block.mjs) and operates in coordination with the Cardano blockchain and the Blockfrost API. It periodically retrieves IPDC block information from the Cardano blockchain and provides the data to the broadcasting station for IPDC transmission.



2.2.2 System Components

2.2.2.1 Block Information Retrieval (Blockfrost API)

- Utilizes the Blockfrost API to fetch the latest Cardano blockchain block information every 15 seconds.
- Each block's data includes:
  - ・Block hash
  - ・Previous block information
  - ・Transaction list within the block
- This information ensures data integrity and consistency across the system.
- Implements a file lock mechanism to pause data retrieval if a specified file exists, preventing conflicts with other processes.

2.2.2.2 IPDC Hash Generation

- Based on the retrieved block data, an IPDC-format hash is generated for each block.
- The sha3_256** algorithm** is used to create the hash by combining:
  - ・The previous block's IPDC hash
  - ・The current block's data
- The generated IPDC hash is referenced by the next block, ensuring data continuity and reliability.

2.2.2.3 File Storage and Trigger File Update

- A certificate file containing the calculated IPDC hash and block information is generated in JSON format and stored for each block. This ensures that each block's integrity is recorded as verified data.
- The certificate file name is appended to the trigger.txt file, making it easier for external systems to detect newly generated certificate files.
- By updating the trigger file, the system supports efficient data retrieval for external integrations.

2.2.2.4 Log Management

- Utilizes the Log4js library to log information on data retrieval, IPDC hash generation, certificate file creation, anomaly detection, and error handling.
- Each log entry contains processing execution status and detailed error information, which helps monitor the tool's operation and facilitates troubleshooting.

2.2.2.5 Automatic Deletion of Old Files

- Certificate files that exceed a specified retention period will be automatically deleted to efficiently manage the system's storage.
- The deletion process is executed once per day, ensuring proper organization of files that do not require long-term storage.

2.2.2.6 Data Flow

- **Retrieving Block Information**
  The latest block information on the Cardano blockchain is retrieved via the Blockfrost API, ensuring consistency with the previously recorded block information.
- **Generating IPDC Hash**
  Based on the IPDC hash of the previous block, a new IPDC hash is calculated using the retrieved block information.
- **Generating and Storing Certificate Files**
  The block information, including the IPDC hash, is stored in a JSON-format certificate file, and the trigger file is updated.
- **Logging and Error Handling**
  Data retrieval, processing results, and error occurrences are recorded in logs.
- **Deleting Old Files**
  Old files that exceed the retention period are regularly deleted to manage storage efficiently.

With this structure, the Certificate IPDC Transmission Support Tool consistently processes everything from retrieving block information to generating the IPDC hash and storing data, ensuring overall data integrity within the system.

2.3 Module Design

This tool consistently performs tasks such as retrieving block information, generating IPDC hashes, saving certificate files, and managing logs within a single module. The roles and processing details of each function are described below.

2.3.1 Module Processing Details

The frontend is responsible for retrieving block information, generating IPDC hashes, creating and saving certificate files, managing logs, and implementing an automatic file deletion feature. The main functions and their roles are as follows:

2.3.1.1 Retrieving Block Information

- **Processing Details**

  The system accesses the Blockfrost API at 15-second intervals to retrieve the latest block information from the Cardano blockchain.

- **Retrieved Data**
  - Latest block hash,
  - Previous block information,
  - Transaction list within the block.

- **File Lock Feature**

  If a specified file exists, the data retrieval process is skipped to prevent conflicts with other processes.

- **Error Handling**

  In case of API request failures, error messages are logged, and the retrieval process is retried in the next cycle.

2.3.1.2 Generating IPDC Hash

- **Processing Details**

  A new IPDC hash is generated by combining the retrieved latest block information with the previously generated IPDC hash.

- **Generation Method**

  The sha3_256 algorithm is used to concatenate the previous IPDC hash with the current block data and compute the new IPDC hash.

  The generated IPDC hash is also used as a reference hash for the next block, ensuring data integrity and continuity.

- **Error Handling**

  If hash generation fails, an error log is recorded, and the process is retried in the next retrieval cycle.

2.3.1.3 Generating and Saving Certificate Files

- **Processing Details**

  The generated IPDC hash and block information are saved as a certificate file in JSON format.

- **File Structure**

  A JSON file is created for each block, containing block height, block hash, transaction list, and IPDC hash.

  The certificate file name is appended to **"trigger.txt"**, enabling external systems to detect file generation.

- **Error Handling**

  If file generation or saving fails, an error log is recorded, and the process is retried in the next cycle.

2.3.1.4 Log Management

- **Processing Details**

  Execution results of each process and error details in case of abnormalities are recorded in logs.

- **Log Content**

  During normal operation, logs record the start and end of processes and successful data retrieval statuses.

  In case of abnormalities, logs store detailed error information, including error messages, location of occurrence, and timestamps.

- **Log Library**

  The **Log4js** library is used to efficiently manage operational information.

2.3.1.5 Automatic Deletion of Old Files

- **Processing Details**

  The system periodically checks the storage directory and deletes certificate files older than a specified number of days.

- **Execution Timing**

  This process runs once per day to automatically delete unnecessary files and maintain efficient storage usage.

- **Error Handling**

  If deletion fails, an error log is recorded, and the process is retried in the next execution cycle.

2.3.1.6 Overall Module Workflow

- **Processing starts every 15 seconds**

  Retrieves block information and generates an IPDC hash.

- **Certificate file generation**

  Creates a certificate file using the retrieved block information and IPDC hash, then updates the trigger file.

- **Logging**

  Records the status of each process and error handling results.

- **Daily automatic deletion**

  Deletes old files exceeding the specified retention period to manage system storage.

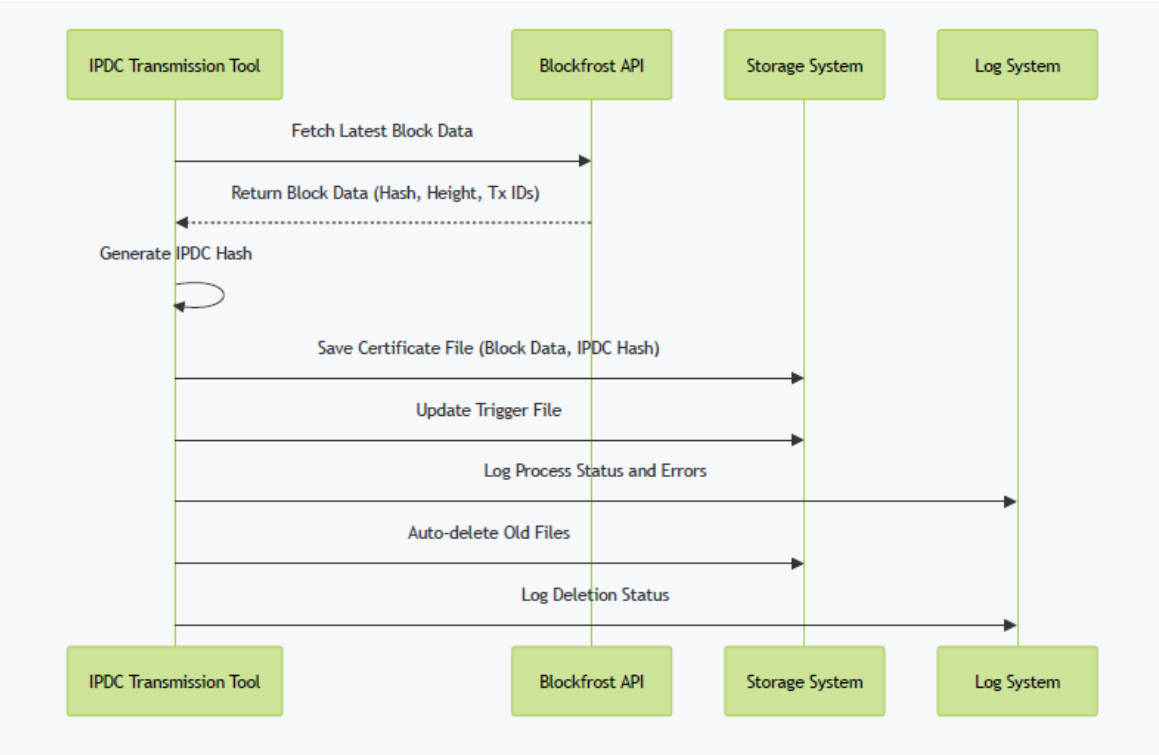
By following this structure, the IPDC Certificate Transmission Support Tool ensures seamless processing from block information retrieval to IPDC hash generation and data storage, maintaining overall data integrity within the system.

2.4 Data Flow and Algorithm Design

2.4.1 Data Flow and Algorithm

The IPDC Certificate Transmission Support Tool performs a series of processes, including retrieving the latest block information, generating an IPDC hash, creating and saving certificate files, and automatically deleting old files. Below, we describe the algorithms used in each step and the overall data flow.

The data flow is illustrated as follows:



2.4.1.1 Retrieving Block Information

- **Objective**

  Retrieve the latest block information from the Cardano blockchain to use as fundamental data for processing within the tool.

- **Algorithm**

  The latest block data is assigned an IPDC-format hash to ensure data integrity and consistency.

- **Error Handling**

  If the API call fails, the error is logged, and the process is retried in the next retrieval cycle.

2.4.1.2 Generating IPDC Hash

- **Objective**

  Use the **CardanoWasm** library to encode the credential certificate data as **Auxiliary Data** and generate a transaction.

- **Algorithm**

  ＜Retrieving Previous IPDC Hash＞

  The previously generated IPDC hash is referenced and combined with the current block data to generate a new hash.

  ＜Hash Generation Method＞

  The sha3_256 algorithm is used to hash the following data:

  - The previous IPDC hash
  - The latest block hash
  - The transaction ID list

＜**Generation Result**＞

The newly computed IPDC hash is stored and referenced in the next cycle.

- **Error Handling**

  If hash generation fails, the error is logged, and the process is retried in the next cycle.

2.4.1.3 **Generating and Saving Certificate Files**

- **Objective**

  Save the computed IPDC hash and block information as a **JSON** file for data verification and integration with external systems.

- **Processing Details**

  The IPDC hash and block information are structured in JSON format, and a certificate file is generated.　The file includes:

  Block height、Block hash、IPDC hash、Transaction ID list

  ＜**Trigger File Update**＞

  The new certificate file name is appended to **"trigger.txt"**, allowing other systems to detect newly generated files.

- **Error Handling**

  If file generation fails, the error is logged, and the process is retried in the next cycle.

2.4.1.4 Automatic Deletion of Old Files

- **Objective**

  Remove certificate files that exceed a predefined retention period to manage system storage efficiently.

- **Processing Details**

  The tool scans the certificate file storage directory and automatically deletes files that have exceeded the specified number of days (e.g., **7 days**).

  This process runs **once per day**, periodically clearing unnecessary data.

- **Error Handling**

  If the deletion process fails, the error is logged, and the process is retried in the next execution cycle.

2.4.2 Algorithm details

2.4.2.1 ブ Block information acquisition algorithm

- **Procedure**
  1. Send a request to the Blockfrost API endpoint at 15-second intervals.
  2. Retrieve the latest block hash, block height, previous block information, and transaction ID list.
  3. Pass the retrieved block information to the next IPDC hash generation process.
- **Error Handling**

  If an error is returned from the Blockfrost API, the system will not retry and will attempt to retrieve the data in the next cycle.

2.4.2.2 IPDC Hash Generation Algorithm

- **Procedure**
  1. Retrieve the IPDC hash generated in the previous cycle.
  2. Combine the latest block hash and the transaction ID list, then generate a hash using the **SHA3-256** algorithm.
  3. Save the generated IPDC hash as input data for the next cycle.
- **Error Handling**

  If hash generation fails, log the error and retry in the next cycle.

2.4.2.3 Certificate File Generation Algorithm

- **Procedure**

  1. Format the IPDC hash and block information into JSON format.
  2. Generate and save a certificate file (e.g., block_<height>.json) based on the JSON data.
  3. Append the new file name to **"trigger.txt"**.

- **Error Handling**

  If file generation fails, log the error and retry in the next cycle.

2.4.2.4 **Old File Deletion Algorithm**

- **Procedure**

  1. Scan the directory where certificate files are stored.
  2. Check the creation date of each file and delete files that exceed the specified retention period.
  3. Execute once per day to automatically remove old data.

- **Error Handling**

  If a file fails to be deleted, log the error and retry during the next execution.

2.4.2 Data Flow Summary

2.4.2.1 Data Flow

1. **Retrieve block information** from the API
2. **Generate the IPDC hash**
3. **Generate and save the certificate file**
4. **Log the process**
5. **Delete old files**

This data flow and algorithm enable the tool to periodically manage the latest block information in **IPDC format**, automate file storage, and handle old data management. Additionally, error handling is integrated into each processing step, ensuring stable operation.

2.5 Data Design

The IPDC Certificate Transmission Support Tool handles various types of data, including block information, IPDC hashes, trigger files, and logs. The following sections describe the role, structure, and storage format of each data type.

2.5.1 Data Items

2.5.1.1 Block Information

- **Description**

  The latest block information retrieved from the Cardano blockchain using the Blockfrost API.

- **Content**

  <Block Hash>

  A hash value used as a unique identifier for the block.

  <Block Height>

  A numeric value indicating the block's position in the blockchain.

  <Transaction ID List>

  A list of all transaction IDs included in the block.

- **Example Data**

```
{
   "hash": "abc123...",
   "height": 123456,
   "previous_block": "def456...",
   "transactions": ["tx1", "tx2", "tx3"]
}
```

2.5.1.2 IPDC Hash

- **Description**

  The IPDC hash assigned to each block is generated by combining the previous block's IPDC hash with the current block data. It is used to ensure data integrity and continuity.

- **Generation Method**

  The IPDC hash is calculated using the following data:

  - Previous IPDC Hash
  - Current Block Hash
  - Transaction ID List

- **Data Format**

  String (Hexadecimal hash value)

- **Example Data**

```
"ipdc_hash": "789abc..."
```

2.5.1.3 Certificate File

- **Description**

  A JSON-formatted file containing block information and the IPDC hash, generated for each block. This file is used for data integration and verification with external systems.

- **Storage Location**

  Saved in a directory specified in the tool settings. The file name includes the block height (e.g., block_<height>.json).

- **Content**

  <block height> The block height at the time of generation.

  <block hash> The hash value of the current block.

  <IPDC hash> The latest IPDC hash.

  <transaction ID List> The list of transaction IDs in the block.

- **Data Format**

  JSON format

- **Example Data**

```
{
   "height": 123456,
   "block_hash": "abc123...",
   "ipdc_hash": "789abc...",
   "transactions": ["tx1", "tx2", "tx3"]
}
```

2.5.1.4 Trigger File (trigger.txt)

- **Description**

  A text file used to notify external systems of newly generated certificate files. Each new certificate file path is appended to this file.

- **Content**

  The file path of the latest generated certificate file is added to the text file.

- **Data Format**

  Text file (.txt)

- **Example Data (trigger.txt)**

```
/path/to/certificate/block_123456.json
/path/to/certificate/block_123457.json
```

2.5.1.5 Log File

- **Description**

  The log file records the execution status, processing results, and error information of the tool. It is used for operation monitoring and troubleshooting.

- **Content**
  - Execution results of normal processing
    (e.g., successful block information retrieval, completion of IPDC hash generation)
  - Error messages
    (e.g., API errors, file save failures)
  - Processing start and end timestamps

- **Data Format**

  Text file (.log format)

- **Example Data**

```
[INFO] 2024-11-01 15:00:00 - Block data fetched successfully for block height 123456
[ERROR] 2024-11-01 15:15:00 - Failed to fetch block data - API Error: 503 Service Unavailable
```

2.5.1.6 Deletion of Old Files

- **Description**

  This process automatically deletes certificate files that have exceeded a predefined retention period to efficiently manage system storage. Files stored in the designated directory that are older than the specified number of days are targeted for deletion.

- **Data Format**

  List of file paths of deleted files

- **Example Data**

  ```
  /path/to/certificate/block_123456.json
  ```

2.5.2 Summary of Data Design

- **Block Information**

  Retrieves the latest block data and serves as the foundation for generating the IPDC hash.

- **IPDC Hash**

  A unique hash assigned to each block, ensuring data integrity and continuity.

- **Certificate File**

  Contains IPDC hash and block information, used for data integration with external systems.

- **Trigger File**

  Notifies external systems about **newly generated certificate files.**

- **Log File**

  Records processing status and error details, aiding operation monitoring and troubleshooting.

- **Deletion of Old Files**

  Periodically removes unnecessary files to maintain efficient storage management.

This **data design** ensures **organized data management within the tool**, defining the **storage format of each data type**, while maintaining **reliability and consistency in processing.**

3. Functionality verification

3.1 Certificate issuing tool

The following report shows the results when the certificate issuance tool was actually executed.

3.1.1 Input Data


```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1"
  ],
  "id": "urn:uuid:12345-67890",
  "type": ["VerifiableCredential", "DisasterReliefVolunteerCredential"],
  "issuer": {
    "id": "https://disaster-response-headquarters.example.org",
    "name": "Disaster Response Headquarters"
  },
  "issuanceDate": "2024-11-03T00:00:00Z",
  "credentialSubject": {
    "id": "did:example:volunteer-001",
    "name": "John Doe",
    "role": "Heavy Machinery Operator",
    "skills": [
      "Excavator Operation",
      "Bulldozer Operation",
      "Crisis Communication"
    ],
```

```
    "certifications": [
      {
        "type": "HeavyMachineryCertification",
        "issuedBy": "Local Training Institute",
        "issueDate": "2023-05-01"
      }
    ],
    "authorizedRegions": [
      {
        "region": "Prefecture A, City B",
        "validUntil": "2024-12-31"
      }
    ]
  }
}
```


Sending and receiving addresses

---

addr_test1qq54kguth0dtagn4j5skz3qwcm7tz2qrj8klrllfpfefe2pftv3chw76h638t9fpv9zqa3huky5q8y0d78l7jznjnj5qscstrm

---

3.1.2 Console output result(Frontend Side)

Signed Transaction (Hex):

84a400818258209d1d5c1505e6ceab4e6821af1cda182c3287ea1446b213cc7b907505106a0a95000181
82583900295b238bbbdabea275952161440ec6fcb1280391edf1ffe90a729ca8295b238bbbdabea27595
2161440ec6fcb1280391edf1ffe90a729ca81b0000000253859d00021a00030d400758209d18dbac3ebd
de6a45a2e4f342b6950eafef5b5fef00022cba5828e1c4f46b73a100818258203745f60982871f6d49d0
eb0cf1b02b4eafd12847993311292387df6ed86fcad5840f4ad182a97c9ca6888985d5bfd735699f93f
abf3a3681787f2fd601064693f38f24d7be14bd84f57c9d21a917511a39df4a2f047c70d6a529745051e
98582405f5a11902d1a66840636f6e7465787481782668747470733a2f2f7777772e77332e6f72672f32
3031382f63726564656e7469616c732f76317163726564656e7469616c5375626a656374a67161757468
6f72697a6564526567696f6e7381a266726567696f6e7450726566656374757265726520412c204369747920
426a76616c6964556e74696c6a323032342d31322d33316e6365727469666696361746 96f6e7381a36969
737375654461746 56a323032332d30352d303168697373756564427978184c6f63616c20547261696e69
6e6720496e737469747574656474797065781b48656176794d616368696e65727943657274696669636 1
74696f6e6269647819646964 3a6578616d706c653a766f6c756e746565722d303031646e616d65684a6f
686e20446f6564726f6c65781848656176792 04d616368696e657279204f70657261746f7266736b696c
6c7383734578636176174 6f72204f7065726174696f6e7342756c6c646f7a6572204f7065726174696f
6e744372697369732043 6f6d6d756e69636174696f6e6269647475726e3a757569643a31323334352d36
373839306c69737375616e6365446174657432303234 2d31312d30335430303a30303a30305a66697373
756572a2626964783268747470733a2f2f6469736173746 5722d726573706f6e73652d68656164717561
72746572732e6578616d706c652e6f7267646e616d65781e446973617374657220526573706f6e736520
486561647175617274657273647479706582745665726966961626c6543726564656e7469616c782144
697361737465725265 6c696566566f6c756e7465657243726564656e7469616c

select mac UTXO address:


```
{
    "address": "addr_test1qq54kguth0dtagn4j5skz3qwcm7tz2qrj8klrllfpfefe2pftv3chw
                76h638t9fpv9zqa3huky5q8y0d78l7jznjnj5qscstrm",
    "tx_hash": "9d1d5c1505e6ceab4e6821af1cda182c3287ea1446b213cc7b907505106a0a95",
    "tx_index": 0,
    "output_index": 0,
    "amount": [
        {
            "unit": "lovelace",
            "quantity": "9991400000"
        }
    ],
    "block": "21a599e2add362e7bba5c2fb5c0b380d844fa5fe4a8b12fa6932a034605e5955",
    "data_hash": null,
    "inline_datum": null,
    "reference_script_hash": null
}
```


Response:


```
{
    "code": "success",
    "data": "3f01c67ba4f0fd985eb22e3fb86fee59e8407adf1c345c2c871f52f97a25a385"
}
```

Block information:


```
{
    "hash": "3f01c67ba4f0fd985eb22e3fb86fee59e8407adf1c345c2c871f52f97a25a385",
    "block": "215a4e8369ddd973a24c6ef964d2e9e515fb417a408fcc63d428fbbbea622012",
    "block_height": 2865317,
    "block_time": 1731041107,
    "slot": 75357907,
    "index": 5,
    "output_amount": [
        {
            "unit": "lovelace",
            "quantity": "9991200000"
        }
    ],
    "fees": "200000",
    "deposit": "0",
    "size": 871,
    "invalid_before": null,
    "invalid_hereafter": null,
    "utxo_count": 2,
    "withdrawal_count": 0,
    "mir_cert_count": 0,
    "delegation_count": 0,
    "stake_cert_count": 0,
    "pool_update_count": 0,
    "pool_retire_count": 0,
    "asset_mint_or_burn_count": 0,
    "redeemer_count": 0,
    "valid_contract": true
}
```

```
Block Transactions:
[
    "859254002797770a56c82708f30a760513f58ae6237927f6ee02338bf79bedd9",
    "6addc5f8afdc91e9874600805fc8a40afcef730a864d7eb2869bb35655e60e22",
    "adc864139990d6fe66d7440142d9d77a1d875897afd7a282b991aeabdbe2c01f",
    "670e1b3b46d7791deb32107b180468aafb0594979c9525c5e977945589d82ce3",
    "e5586f8dbe7a4c87df2355d4b2baf0947389cc50701eee43445e3c66aae04bcd",
    "3f01c67ba4f0fd985eb22e3fb86fee59e8407adf1c345c2c871f52f97a25a385",
    "af0013afcfa12401d899da580e6031446c330f4a4d4e707a57bdd5cbcb6d0c60"
]

Transaction Hashes (SHA3-256):
[
    "cf81a49d51b616b9badb35b49b43637fe48e7e6e68fcb368e95797fa8208ba2e",
    "f9df2119ece0cb17a9ba50252ac8041719a45eaf35299412830354655187258 6",
    "c33051e15854102e6dc02cf4f8ac4c6320807331e73de3947fb2bcce06cfd6ca",
    "816ca477bedca3651df77494faf041c87d66598d1ab9b45dd96c7b4a17f3c1a0",
    "e6313c311988f179d32640164472c97347a071e57ca4b2a4536e29e307cffe5e",
    "447bca38f9315b7c1ee6d02ccf87da49fb4982867a92d911d3761959d01aa693",
    "565823c4dc90c40314733c9e9ca89c776c5b945603495a71aea487446278fd4d"
]

Merkle Tree:
[
    "cf81a49d51b616b9badb35b49b43637fe48e7e6e68fcb368e95797fa8208ba2e",
    "f9df2119ece0cb17a9ba50252ac8041719a45eaf35299412830354655187258 6",
    "c33051e15854102e6dc02cf4f8ac4c6320807331e73de3947fb2bcce06cfd6ca",
    "816ca477bedca3651df77494faf041c87d66598d1ab9b45dd96c7b4a17f3c1a0",
    "e6313c311988f179d32640164472c97347a071e57ca4b2a4536e29e307cffe5e",
    "447bca38f9315b7c1ee6d02ccf87da49fb4982867a92d911d3761959d01aa693",
    "565823c4dc90c40314733c9e9ca89c776c5b945603495a71aea487446278fd4d"
]
```

```
Merkle Path:
{
    "position": "left",
    "hash": "0b455a355919390bfc61082f3bdc339a402feec2184bb5e1cb66fe1bc2f3b699"
}

Is the leaf hash valid in the Merkle tree? True
```

3.1.3 Console output result(Backend Side)

```
<Start time>

[ipdc@ip-172-31-3-109 ipdc]$ node cardano_server.mjs
[2024-11-08T13:44:30.230] [INFO] system - cardano_server.mjs start

<Receive time>

[2024-11-08T13:44:45.154] [INFO] system - ::ffff:92.203.104.75 -
- "OPTIONS /submit HTTP/1.1" 200 13 "" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, lik e Gecko) Chrome/130.0.0.0 Safari/537.36"
84a400818258209d1d5c1505e6ceab4e6821af1cda182c3287ea1446b213cc7b907505106a0a9500018
182583900295b238bbbdabea275952161440ec6fcb1280391edf1ffe90a729ca8295b238bbbdabea275
952161440ec6fcb1280391edf1ffe90a729ca81b0000000253859d00021a00030d400758209d18dbac3
ebdde6a45a2e4f342b6950eafef5b5fef00022cba5828e1c4f46b73a100818258203745f60982871f6d
49d0eb0cf1b02b4eafd128479933311292387df6ed86fcad5840f4ad182a97c9ca6888985d5bfd73569
9f93fabf3a3681787f2fd601064693f38f24d7be14bd84f57c9d21a917511a39df4a2f047c70d6a5297
45051e98582405f5a11902d1a66840636f6e7465787481782668747470733a2f2f7777772e77332e6f7
2672f323031382f63726564656e7469616c732f76317163726564656e7469616c5375626a656374a671
617574686f72697a6564526567696f6e7381a266726567696f6e7450726566656374757265726520412c204
369747920426a76616c6964556e74696c6a323032342d31322d33316e6365727469666696361746 96f6e7
381a3696973737565446174656a323032332d30352d303168697373756564427978184c6f63616c20547
261696e696e6720496e737469747574656474797065781b48656176794d616368696e657279436572746
96669636174696f6e6269647819646964 3a6578616d706c653a766f6c756e746565722d303031646e616
d65684a6f686e20446f6564726f6c657818486561767920 4d616368696e657279204f70657261746f726
6736b696c6c7383734578636176174 6f72204f7065726174696f6e7342756c6c646f7a6572204f70657
26174696f6e744372697369732043 6f6d6d756e69636174696f6e6269647475726e3a757569643a31323
334352d36373839306c69737375616e6365446174657432303234 2d31312d30335430303a30303a30305
a66697373756572a2626964783268747470733a2f2f6469736173746572 2d726573706f6e73652d68656
164717561727465726f6e6578616d706c652e6f7267646e616d65781e4469736173746572205265737 06
f6e736 5204865616471756172746572736474797065827456657269666961626c6543726564656e7469
616c782144697361737465725265 6c696566566f6c756e746565724372656465 6e7469616c
```

```
Transaction sent successfully: {
  status: 200,
  statusText: 'OK',
  headers: Object [AxiosHeaders] {
    date: 'Fri, 08 Nov 2024 04:44:45 GMT',
    'content-type': 'application/json;charset=utf-8',
    'transfer-encoding': 'chunked',
    connection: 'close',
    vary: 'Origin',
        'access-control-allow-origin': '*',
    'bf-submit-server':
        'e1f4ccc29ee5a9bb1f2d55e35e58119727c3f3f0e0da9c0cf0a9582299058083',
    'cf-cache-status': 'DYNAMIC',
    'report-to': '{"endpoints":
        [{"url":"https:¥¥/¥¥/a.nel.cloudflare.com¥¥/report¥¥/v4?s=Ix24pfKCKhf7kaJaDIAn
        FOJtmIAGs9X76AoseLb6Fibqf%2B1odiBlvKL6eYy8YwdNorv6bdGJcBEVOHGrOXqnR%2Fh6FfhMjYCi
        ZYRYzNmOaVSS9rzk9DwOulXrX1SYxAHgAqfOgsAmeCyBXOCKoFTC"}],"group":"cf-nel",
        "max_age":604800}',
    nel: '{"success_fraction":0,"report_to":"cf-nel","max_age":604800}',
    server: 'cloudflare',
    'cf-ray': '8df2e8de98d9d753-NRT',
    'server-timing':
         'cfL4;desc="?proto=TCP&rtt=2207&sent=4&recv=5&lost=0&retrans=0&sent_bytes=2340&
          recv_bytes=1668&delivery_rate=1295749&cwnd=252&unsent_bytes=0&cid=b787176dfb3
          348e0&ts=673&x=0"'
  },
  config: {
    transitional: {
      silentJSONParsing: true,
      forcedJSONParsing: true,
      clarifyTimeoutError: false
    },
    adapter: [ 'xhr', 'http', 'fetch' ],
    transformRequest: [ [Function: transformRequest] ],
```

```
    transformResponse: [ [Function: transformResponse] ],
    timeout: 0,
    xsrfCookieName: 'XSRF-TOKEN',
    xsrfHeaderName: 'X-XSRF-TOKEN',
    maxContentLength: -1,
    maxBodyLength: -1,
    env: { FormData: [Function], Blob: null },
    validateStatus: [Function: validateStatus],
    headers: Object [AxiosHeaders] {
      Accept: 'application/json, text/plain, */*',
      'Content-Type': 'application/cbor',
      project_id: 'preprod7DHcGID2pFZxqgFSjL84JXRrxCS7Cblf',
      'User-Agent': 'axios/1.7.7',
      'Content-Length': '872',
      'Accept-Encoding': 'gzip, compress, deflate, br'
    },
    method: 'post',
    url: 'https://cardano-preprod.blockfrost.io/api/v0/tx/submit',
    data: <Buffer 84 a4 00 81 82 58 20 9d 1d 5c 15 05 e6 ce ab 4e 68 21 af 1c da
    18 2c 32 87 ea 14 46 b2 13 cc 7b 90 75 05 10 6a 0a 95 00 01 81 82 58 39 00
    29 5b 23 8b ... 822 m ore bytes>
  },
  request: <ref *1> ClientRequest {
    _events: [Object: null prototype] {
      abort: [Function (anonymous)],
      aborted: [Function (anonymous)],
      connect: [Function (anonymous)],
      error: [Function (anonymous)],
      socket: [Function (anonymous)],
      timeout: [Function (anonymous)],
      finish: [Function: requestOnFinish]
    },
    _eventsCount: 7,
    _maxListeners: undefined,
```

```
  outputData: [],
  outputSize: 0,
  writable: true,
  destroyed: true,
  _last: true,
  chunkedEncoding: false,
  shouldKeepAlive: false,
  maxRequestsOnConnectionReached: false,
  _defaultKeepAlive: true,
  useChunkedEncodingByDefault: true,
  sendDate: false,
  _removedConnection: false,
  _removedContLen: false,
  _removedTE: false,
  strictContentLength: false,
  _contentLength: '872',
  _hasBody: true,
  _trailer: '',
  finished: true,
  _headerSent: true,
  _closed: true,
  socket: TLSSocket {
    _tlsOptions: [Object],
    _secureEstablished: true,
    _securePending: false,
    _newSessionPending: false,
    _controlReleased: true,
    secureConnecting: false,
    _SNICallback: null,
    servername: 'cardano-preprod.blockfrost.io',
    alpnProtocol: false,
    authorized: true,
    authorizationError: null,
    encrypted: true,
```

```
  _events: [Object: null prototype],
  _eventsCount: 9,
  connecting: false,
  _hadError: false,
  _parent: null,
  _host: 'cardano-preprod.blockfrost.io',
  _closeAfterHandlingError: false,
  _readableState: [ReadableState],
  _maxListeners: undefined,
  _writableState: [WritableState],
  allowHalfOpen: false,
  _sockname: null,
  _pendingData: null,
  _pendingEncoding: '',
  server: undefined,
  _server: null,
  ssl: null,
  _requestCert: true,
  _rejectUnauthorized: true,
  parser: null,
  _httpMessage: [Circular *1],
  write: [Function: writeAfterFIN],
  [Symbol(res)]: null,
  [Symbol(verified)]: true,
  [Symbol(pendingSession)]: null,
  [Symbol(async_id_symbol)]: 315,
  [Symbol(kHandle)]: null,
  [Symbol(lastWriteQueueSize)]: 0,
  [Symbol(timeout)]: null,
  [Symbol(kBuffer)]: null,
  [Symbol(kBufferCb)]: null,
  [Symbol(kBufferGen)]: null,
  [Symbol(kCapture)]: false,
  [Symbol(kSetNoDelay)]: false,
```

```
    [Symbol(kSetKeepAlive)]: true,
    [Symbol(kSetKeepAliveInitialDelay)]: 60,
    [Symbol(kBytesRead)]: 968,
    [Symbol(kBytesWritten)]: 1183,
    [Symbol(connect-options)]: [Object],
    [Symbol(RequestTimeout)]: undefined
  },
  _header: 'POST /api/v0/tx/submit HTTP/1.1¥r¥n' +
    'Accept: application/json, text/plain, */*¥r¥n' +
    'Content-Type: application/cbor¥r¥n' +
    'project_id: preprod7DHcGID2pFZxqgFSjL84JXRrxCS7Cblf¥r¥n' +
    'User-Agent: axios/1.7.7¥r¥n' +
    'Content-Length: 872¥r¥n' +
    'Accept-Encoding: gzip, compress, deflate, br¥r¥n' +
    'Host: cardano-preprod.blockfrost.io¥r¥n' +
    'Connection: close¥r¥n' +
    '¥r¥n',
  _keepAliveTimeout: 0,
  _onPendingData: [Function: nop],
  agent: Agent {
    _events: [Object: null prototype],
    _eventsCount: 2,
    _maxListeners: undefined,
    defaultPort: 443,
    protocol: 'https:',
    options: [Object: null prototype],
    requests: [Object: null prototype] {},
    sockets: [Object: null prototype] {},
    freeSockets: [Object: null prototype] {},
    keepAliveMsecs: 1000,
    keepAlive: false,
    maxSockets: Infinity,
    maxFreeSockets: 256,
    scheduling: 'lifo',
```

```
    maxTotalSockets: Infinity,
    totalSocketCount: 0,
    maxCachedSessions: 100,
    _sessionCache: [Object],
    [Symbol(kCapture)]: false
  },
  socketPath: undefined,
  method: 'POST',
  maxHeaderSize: undefined,
  insecureHTTPParser: undefined,
  path: '/api/v0/tx/submit',
  _ended: true,
  res: IncomingMessage {
    _readableState: [ReadableState],
    _events: [Object: null prototype],
    _eventsCount: 4,
    _maxListeners: undefined,
    socket: [TLSSocket],
    httpVersionMajor: 1,
    httpVersionMinor: 1,
    httpVersion: '1.1',
    complete: true,
    rawHeaders: [Array],
    rawTrailers: [],
    aborted: false,
    upgrade: false,
    url: '',
    method: null,
    statusCode: 200,
    statusMessage: 'OK',
    client: [TLSSocket],
    _consuming: true,
    _dumped: false,
    req: [Circular *1],
```

```
    responseUrl: 'https://cardano-preprod.blockfrost.io/api/v0/tx/submit',
    redirects: [],
    [Symbol(kCapture)]: false,
    [Symbol(kHeaders)]: [Object],
    [Symbol(kHeadersCount)]: 28,
    [Symbol(kTrailers)]: null,
    [Symbol(kTrailersCount)]: 0,
    [Symbol(RequestTimeout)]: undefined
  },
  aborted: false,
  timeoutCb: null,
  upgradeOrConnect: false,
  parser: null,
  maxHeadersCount: null,
  reusedSocket: false,
  host: 'cardano-preprod.blockfrost.io',
  protocol: 'https:',
  _redirectable: Writable {
    _writableState: [WritableState],
    _events: [Object: null prototype],
    _eventsCount: 3,
    _maxListeners: undefined,
    _options: [Object],
    _ended: true,
    _ending: true,
    _redirectCount: 0,
    _redirects: [],
    _requestBodyLength: 872,
    _requestBodyBuffers: [],
    _onNativeResponse: [Function (anonymous)],
    _currentRequest: [Circular *1],
    _currentUrl: 'https://cardano-preprod.blockfrost.io/api/v0/tx/submit',
    [Symbol(kCapture)]: false
  },
```

```
    [Symbol(kCapture)]: false,
    [Symbol(kBytesWritten)]: 0,
    [Symbol(kEndCalled)]: true,
    [Symbol(kNeedDrain)]: false,
    [Symbol(corked)]: 0,
    [Symbol(kOutHeaders)]: [Object: null prototype] {
      accept: [Array],
      'content-type': [Array],
      project_id: [Array],
      'user-agent': [Array],
      'content-length': [Array],
      'accept-encoding': [Array],
      host: [Array]
    },
    [Symbol(kUniqueHeaders)]: null
  },
  data: '3f01c67ba4f0fd985eb22e3fb86fee59e8407adf1c345c2c871f52f97a25a385'
}
{
  code: 'success',
  data: '3f01c67ba4f0fd985eb22e3fb86fee59e8407adf1c345c2c871f52f97a25a385'
}

[INFO] system - ::ffff:92.203.104.75 - - "POST /submit HTTP/1.1" 200 92 "" "Mozilla/5.0
(Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like G ecko) Chrome/130.0.0.0
Safari/537.36"
```

3.2 IPDC Transmission Support Tool

The log when the IPDC transmission support tool was actually run is shown below.

3.2.1 Node Monitoring

```
### ログ出力  height: 2865315 - 2865317
[INFO] system - - prevHash:
ea315543d5d1200153201a88caca3ebcb313ff179ec3716db08bb757dbcc1f5a
[INFO] system - - height: 2865315
[INFO] system - - calcHash:
191f82fc9a796ae69b47198413bd02e0d3bad4856b89ccaa4a5bcd256145ade3
[INFO] system - = prevIpdcHash:
68bbb96f766f7652d00eda37942d6142553d1dcfcf0f44f043952ca46c1efe85
[INFO] system - = txsHash: b61cdfc12f6bae863a37f833cfca166e457eeb0b6b589462455d6736479b425a
[INFO] system - = calcIpdcHash:
4b01c979e64484d8fded9ad375079e4d1ea77b9bd2a0a6cf49956110f2e49859
[INFO] system - writeFileWithRetry cardano_data/cardano_head_2865315.json
[INFO] system - push to trigger.txt cardano_head_2865315.json
[INFO] system - skip: 2865315
[INFO] system - transactions count 1
[INFO] system - Transaction IDs:
[ '2e722e8897220d056190709468b3f76c26bc3a054b27ea69ac9da0abe6977ba6' ]
[INFO] system - - prevHash:
191f82fc9a796ae69b47198413bd02e0d3bad4856b89ccaa4a5bcd256145ade3
[INFO] system - - height: 2865316
[INFO] system - - calcHash:
52c4ab7b072b14f29d5b0a915fae42953614a0054d2d491f5af512182a9593f8
[INFO] system - = prevIpdcHash:
4b01c979e64484d8fded9ad375079e4d1ea77b9bd2a0a6cf49956110f2e49859
[INFO] system - = txsHash: cf15cd6f6a993d0175fbbd3a9978ef0d6096bedab970a51455c27b05cbf60d3d
```

```
[INFO] system - = calcIpdcHash:
c93a8d95e1eb2aac138aa2766ad9648884077c4e21c914ae225abd9b0c759825
[INFO] system - writeFileWithRetry cardano_data/cardano_head_2865316.json
[INFO] system - push to trigger.txt cardano_head_2865316.json
[INFO] system - skip: 2865316
[INFO] system - skip: 2865316
[INFO] system - skip: 2865316
[INFO] system - transactions count 7



[INFO] system - Transaction IDs: [
  '859254002797770a56c82708f30a760513f58ae6237927f6ee02338bf79bedd9',
  '6addc5f8afdc91e9874600805fc8a40afcef730a864d7eb2869bb35655e60e22',
  'adc864139990d6fe66d7440142d9d77a1d875897afd7a282b991aeabdbe2c01f',
  '670e1b3b46d7791deb32107b180468aafb0594979c9525c5e977945589d82ce3',
  'e5586f8dbe7a4c87df2355d4b2baf0947389cc50701eee43445e3c66aae04bcd',
  '3f01c67ba4f0fd985eb22e3fb86fee59e8407adf1c345c2c871f52f97a25a385',
  'af0013afcfa12401d899da580e6031446c330f4a4d4e707a57bdd5cbcb6d0c60'
]
[INFO] system - - prevHash:
52c4ab7b072b14f29d5b0a915fae42953614a0054d2d491f5af512182a9593f8
[INFO] system - - height: 2865317
[INFO] system - - calcHash:
215a4e8369ddd973a24c6ef964d2e9e515fb417a408fcc63d428fbbbea622012
[INFO] system - = prevIpdcHash:
c93a8d95e1eb2aac138aa2766ad9648884077c4e21c914ae225abd9b0c759825
[INFO] system - = txsHash: ffa2e67375ca6cd52a8f36ea03d82f12c4590ce0e0448cdbaf61afe908df6519
[INFO] system - = calcIpdcHash:
c7f56556031f17e3d210d14efd8fd1c2d9831f89642e7af762720b48159d76d0
[INFO] system - writeFileWithRetry cardano_data/cardano_head_2865317.json
[INFO] system - push to trigger.txt cardano_head_2865317.json
[INFO] system - skip: 2865317
[INFO] system - skip: 2865317
[INFO] system - transactions count 6
```

```
[INFO] system - Transaction IDs: [
  '2f61029f9bda80e7432bc4aab6ac4399e5fd8f5e79acecb7c42221285678bc34',
  'd3c7c5b78f1d609464de0d9fe2a306d4942cfce4316bf215bd36f50f8e182d59',
  '29ba85a7e33ae684b8782087994519347ff1aef39e80e65a81d88a4071b24c2f',
  '3d0826c86c5da9992618b371d45b69c65e085b4850b5dc9b1f4ac41138efc833',
  'e2910acf846d4de0992360056594f1cef5bbe355cdb8b15d77f13ac966f43f06',
  'd63698d33c32ac27722831b8299a418926933a4451d54b7e714ab43e7adeb6ef'
}
```

End.