



## SDG BLOCKCHAIN ACCELERATOR

**Technical Architecture Document – Plastiks**

## 1. Project Information

- **Project Name:** \_\_\_\_\_Plastiks Main\_\_\_\_\_
- **Challenge & UNDP Office:** \_\_\_\_\_Udai Solanki\_\_\_\_\_
- **Document Version:** \_\_\_\_\_V1\_\_\_\_\_

## 2. Overview

### Overview of Cardano

The Plastik Smart Contract Ecosystem is deployed on Cardano, leveraging its latest Conway Era advancements for improved scalability and smart contract execution. The Conway Era introduces several innovations beneficial to this ecosystem:

- Plutus V2 Enhancements: Provides more efficient scripting, reducing transaction size and execution costs.
- UTxO and Reference Scripts: Enables optimized contract execution, reducing onchain storage needs and improving performance.
- Stake-Based Certification: Strengthens the verification of plastic recovery projects by integrating stake-based voting mechanisms.

The document ensures a scalable, efficient, and decentralized approach to plastic recovery incentives, NFT-based sustainability credits, and transparency. This document provides a comprehensive technical breakdown of the Plastik Smart Contract Ecosystem on Cardano, including:

- Smart Contract Design: A modular architecture using Plutus V2 and the UTxO model.
- Security Mechanisms: Implementation of multi-signature approvals, formal verification, and commit-reveal schemes.
- Blockchain Integration: Use of reference scripts, inline datums, and CIP-68 metadata.
- Risk Mitigation: Identification and resolution strategies for contract vulnerabilities.
- Deployment & Validation: Simulation of security risks and real-time contract monitoring.

The document ensures a scalable, efficient, and decentralized approach to plastic recovery incentives, NFT-based sustainability credits transparency. This provides a comprehensive overview of the Plastik Smart Contract Ecosystem on Cardano, detailing its integration approaches, technical components,

security mechanisms, potential risks, and mitigation strategies to ensure seamless development and deployment.

## 2. Integration Approaches

### 2.1 Smart Contract Architecture

- Plutus-based Smart Contracts: All contracts are written in Plutus V2, leveraging UTXO-based transaction validation.
- Core Contracts:
  - NFTStoreFrontV4: Handles NFT minting, sales, including lazy minting mechanisms.
  - PlastikCryptoV2: Ensures cryptographic signature verification structured signing.
  - PlastikBurner: Implements automatic token burning i.e. discarding the tokens to maintain token economy same process followed in existing system.
  - PlastikNFTV4 : Manages NFT issuance, ownership transfers.
  - PlastikPRGV3: Issues environmental impact NFTs based on verified recycling contributions.
  - PlasticRecoveryProjects: Maintains a whitelisted registry of verified plastic recovery initiatives.
  - VerifiedAccounts: Ensures KYC-verified users interact with core contracts.
  - PlastikRoleV2: Implements multi-signature role-based access control (RBAC) to prevent unauthorized contract modifications.
  - UtilsV2: Provides cryptographic utilities, structured hashing, and reusable helper functions.

Functions.

APIs & Integrations

- Web3 Wallet Integration: Support for Cardano-compatible wallets including Nami, Eternl, and Yoroi, enabling user authentication and transaction signing.
- Sustainability Data Sources: APIs for verified external sustainability data providers to enhance reporting and validation.
- Cardano Blockchain Explorer APIs: Real-time integration with blockchain explorers for tracking and verifying issued plastic and CO2 credits.
- Payment Gateway Integration: Stripe is integrated into the platform, allowing users to make payments via credit or debit card.

- Reporting & Analytics Tools: Integration with third-party visualization tools for generating detailed sustainability impact reports.

## Blockchain Implementation

Blockchain: Cardano (Native Implementation) for issuing and verifying plastic credits as NFTs.

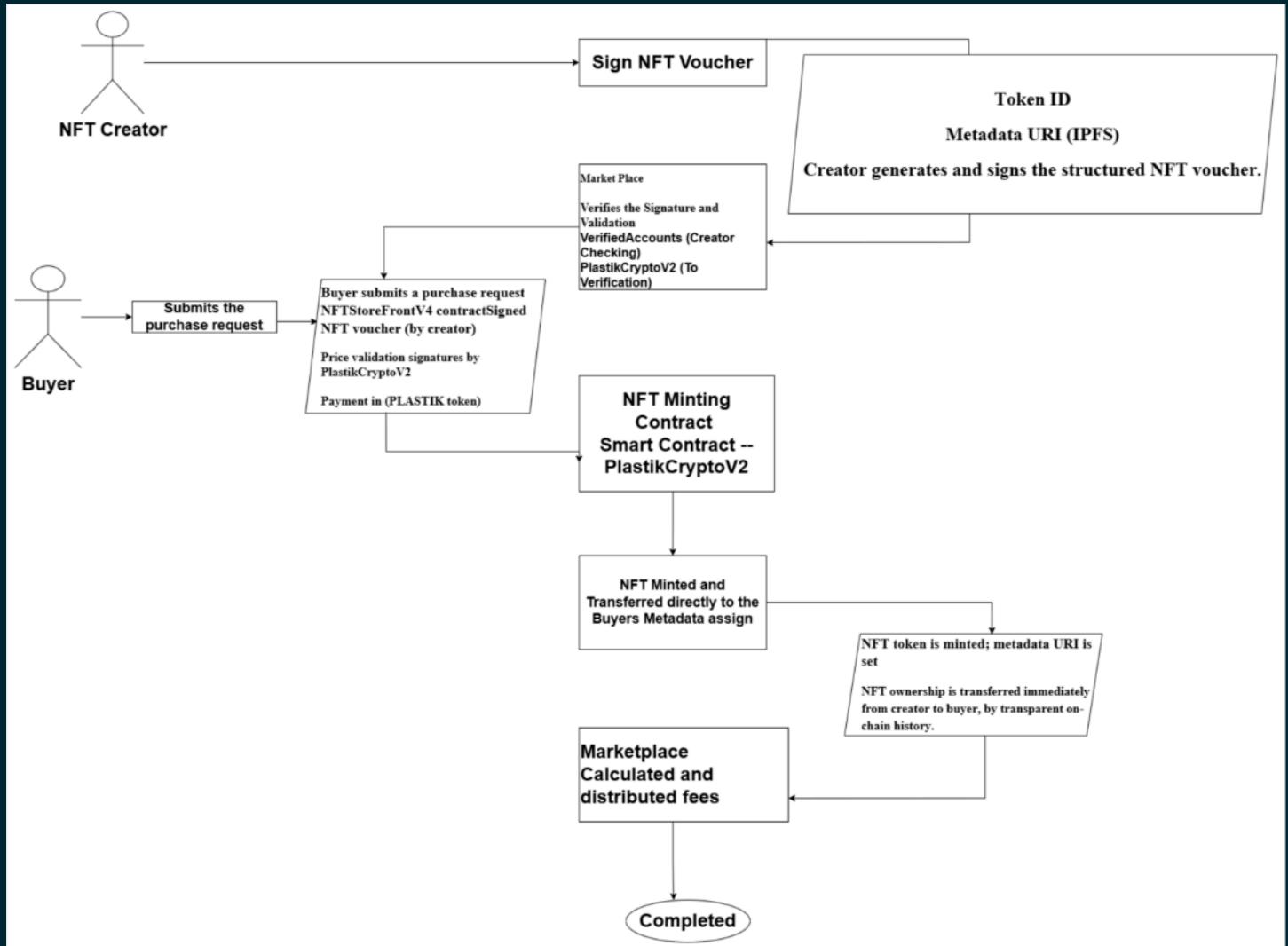
- Smart Contracts:
  - Plutus scripts for executing on-chain logic related to plastic credit issuance and validation.
  - Monetary Policy Scripts for defining the NFT minting rules, ensuring compliance with sustainability verification standards.
- Tokenization:
  - Plastic credits will be issued as NFTs on Cardano, embedding immutable metadata that includes both plastic recovery data and CO2 savings information.
  - Each NFT will contain structured metadata stored on IPFS, linking it to its respective sustainability data.

## Smart Contract Specifications: Libraries to be incorporated in the Haskell File:

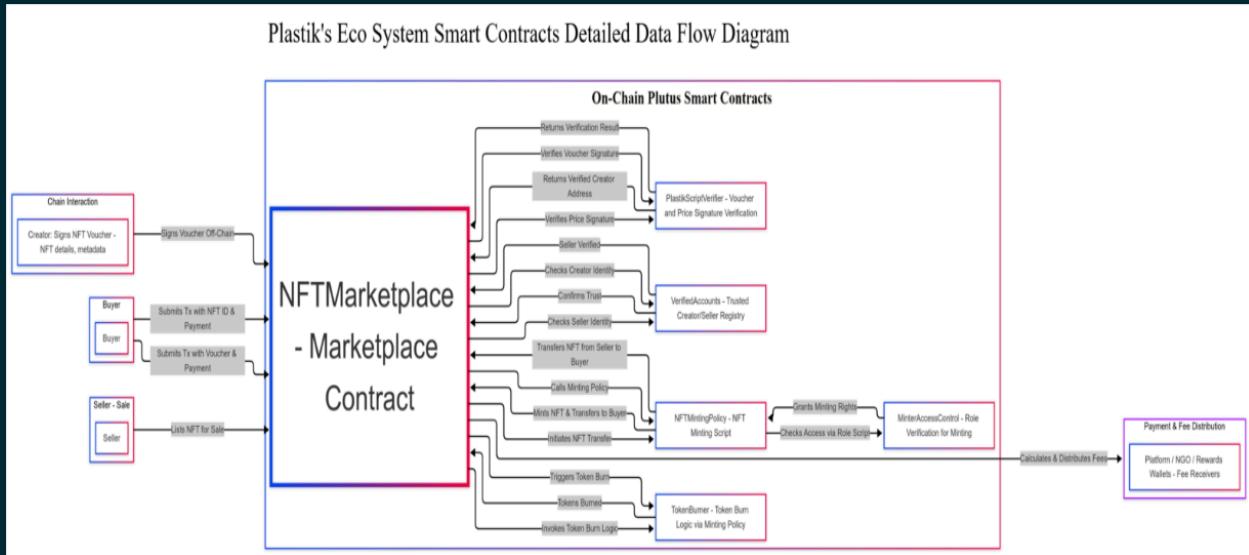
| Libraries                        | Purpose  | Key Features                            |
|----------------------------------|--|---|
| <b>Plutus.V2.Ledger.Api</b>      | <b>Core data types and logic for Plutus scripts</b>  | <b>Validator, Datum, Redeemer</b>       |
| <b>Plutus.V2.Ledger.Contexts</b> | <b>Access transaction context for validation</b>   | <b>ScriptContext, TxInfo</b>            |
| <b>Plutus.V2.Ledger.Scripts</b>  | <b>Compile and manage validator scripts</b>  | <b>mkValidatorScript, Validator</b>     |
| <b>PlutusTx</b>                  | <b>Compile Haskell code to Plutus Core and manage data types</b>                                     | <b>compile, unstableMkElsData</b>       |
| <b>PlutusTx.Prelude</b>          | <b>Provides arithmetic, logical operators, and debugging functions for Plutus Core compatibility</b> | <b>traceIfFalse, (+), div</b>           |
| <b>Ledger</b>                    | <b>Interact with Cardano ledger, manage addresses and hashes</b>                                     | <b>pubKeyHash, unPaymentPubKeyHash</b>  |
| <b>Ledger.Typed.Scripts</b>      | <b>Ensures type safety for smart contracts</b>   | <b>TypedValidator, mkTypedValidator</b> |

### 3. System Architecture Diagram

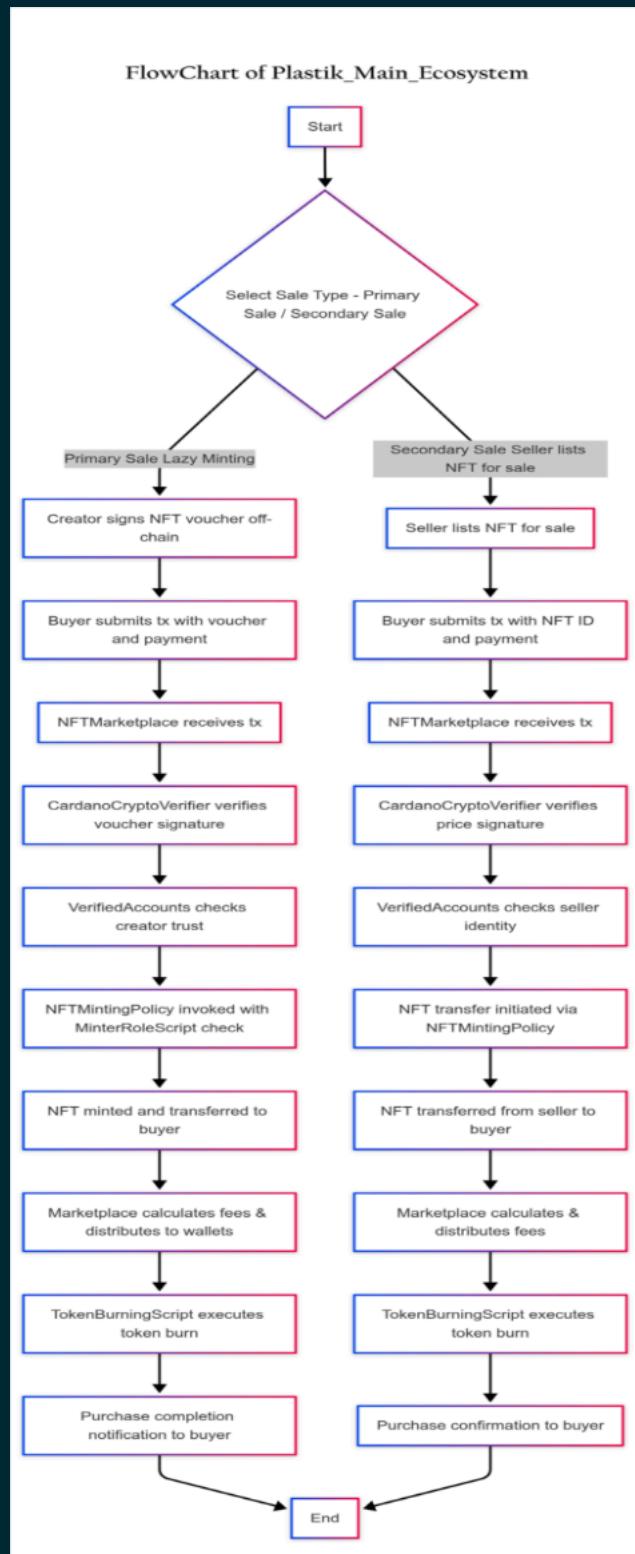
*System-Level Data Flow Diagram (DFD)*



## Smart Contract Execution Flow



## Plastik Ecosystem Flowchart



## Cardano Microservice for Plastiks Marketplace

We have designed a REST-based microservice to abstract blockchain interactions and simplify integration with the marketplace. This document details the structure, API endpoints, data models, and core considerations for building this microservice.

### Objectives

The goals for this microservice include:

- Ensuring a seamless transition of Plastic Credits (NFTs) to Cardano.
- Abstracting the complexities of Cardano's blockchain interactions.
- Maintaining existing marketplace functionalities with minimal disruption.
- Providing clear and intuitive RESTful APIs.

### Technology Stack

For implementing this microservice, we propose:

- Node.js (Express.js)
- Cardano SDK (Blockfrost API, Cardano Serialization Library)
- Docker (Containerization)
- Kubernetes (Google Kubernetes Engine - GKE)
- Google Cloud Platform (GCP) for infrastructure

### REST API Endpoints

Here's a refined version of the REST API documentation, clearly defining endpoints and payload structures aligned. This version explicitly shows that the microservice will receive complete data structures (such as vouchers and sell requests) rather than IDs, given it won't have direct access to the database.

### Wallet Integration & Connectivity:

#### Browser Wallets

Users can connect their Cardano wallets directly through browser extensions.

#### Supported Wallets:

- Eternl

- Lace
- Yoroi

Security: Wallet connection is read-only until the user initiates a transaction.

## 4. Blockchain Design

### Smart Contract Documentation

A detailed breakdown of each Plutus smart contract, including its scope, data structures, and validation logic.

#### 1. Plastik PRG V3 On-Chain Validator (PlastikPRGV3OnChain.hs)

##### 1.1. Scope

The Plastik PRG V3 smart contract is designed to manage a pre-approved redemption system for a specific asset. It establishes an admin who controls the contract's state and a pre-approved public key hash (prePKH) that is authorized to redeem from the contract. The contract also tracks sales data, allowing the admin to add or update sales records associated with different public key hashes.

##### 1.2. Implementation

This is a Plutus V2 validator script parameterized by an admin's public key hash, which is embedded into the script at compile time. It defines two primary actions:

PreRedeem: Allows a pre-approved user to consume the contract's UTxO.

AddOrUpdateSale: Allows the admin to modify the sales data stored in the datum.

##### 1.3. Data Structure: Datum

```
data PlastikDatum = PlastikDatum
  { prePKH      :: PubKeyHash
  , totalSupply :: Integer
  , salesMap    :: [(PubKeyHash, Integer)]
  , assetIdentifier :: BuiltinByteString
  }
```

prePKH: The PubKeyHash of the user who is pre-approved to redeem from the contract.

totalSupply: An integer representing the total supply of the associated asset.

salesMap: A list of pairs, mapping user PubKeyHashes to their corresponding sales amounts.

assetIdentifier: A unique identifier for the asset being managed.

#### 1.4. Data Structure: Redeemer

```
data PlastikRedeemer  
  
= PreRedeem { redeemerPKH :: PubKeyHash }  
  
| AddOrUpdateSale { salePKH :: PubKeyHash, saleAmount :: Integer }
```

PreRedeem: An action to redeem the UTxO, which requires the redeemer to provide their own PubKeyHash.

AddOrUpdateSale: An action for the admin to add a new sale record or update an existing one for a given PubKeyHash (salePKH).

Explanation: This function acts as the central router for validation. It takes the admin's PubKeyHash as a parameter and pattern matches on the redeemer.

For PreRedeem, it enforces a dual-signature requirement from both the pre-approved user and the admin.

For AddOrUpdateSale, it ensures admin control and state consistency by calling checkOutputDatum, which verifies that the state transition is valid and only the salesMap is modified as intended.

## 2. Plastik Role V2 Validator (PlastikRoleV2.hs)

### 2.1. Scope

The Plastik Role V2 contract establishes a simple on-chain role management system. It maintains a list of authorized "minters" controlled by a single admin. The admin has the exclusive right to grant the minter role to new users. The contract can also be used to verify if a given user currently holds the minter role.

### 2.2. Implementation

This Plutus V2 validator script is parameterized by an admin's PubKeyHash. It manages a list of PubKeyHashes in its datum.

The admin can add new PubKeyHashes to the minter list.

Anyone can use the VerifyMinter redeemer to check if a user is authorized, effectively allowing other smart contracts or off-chain logic to query for permissions.

### 2.3. Data Structure: Datum

```
data RoleDatum = RoleDatum { minters :: [PubKeyHash] }
```

minters: A list of PubKeyHashes representing the users who have been granted the minter role.

#### 2.4. Data Structure: Redeemer

```
data RoleRedeemer = GrantMinter PubKeyHash | VerifyMinter PubKeyHash
```

GrantMinter: An action to add a new PubKeyHash to the minters list.

VerifyMinter: An action to check if a PubKeyHash is currently in the minters list.

Explanation: This function contains the core logic. For GrantMinter, it enforces three strict conditions: admin signature, the uniqueness of the new minter, and the correct update of the state. For VerifyMinter, it performs a simple but crucial check to see if the public key is an authorized minter, enabling permissioned actions.

### 3. Plastik Recovery Projects Validator (PlastikRecoveryProjects.hs)

#### 3.1. Scope

This smart contract serves as an on-chain registry for "Plastic Recovery Projects." It is managed by an admin who can add new projects or update the status of existing ones. Each project is identified by a PubKeyHash and has a boolean status (True for active/verified, False for inactive/unverified). The contract allows spending a UTxO only if a specific project is verified.

#### 3.2. Implementation

This is a Plutus V2 validator parameterized by an admin PubKeyHash. The state, a map of projects to their verification status, is stored in the datum.

Admin Actions: The admin can update the project list using the UpdateProject redeemer.

Verification: The CheckProject redeemer allows a transaction to succeed only if the specified project is active, effectively making the contract's UTxO spendable based on a project's on-chain status.

#### 3.3. Data Structure: Datum

```
newtype PRPState = PRPState
```

```
{ projects :: Map PubKeyHash Bool }
```

projects: A map where keys are the PubKeyHash of the recovery projects and values are booleans representing their verification status.

### 3.4. Data Structure: Redeemer

```
data PRPAction =  
    UpdateProject PubKeyHash Bool  
    | CheckProject PubKeyHash
```

UpdateProject: An action to add or modify a project's status. Requires the project's PubKeyHash and its new boolean status.

CheckProject: An action to verify if a project is currently active (True).

Explanation: The mkValidator function enforces the contract's rules. The UpdateProject branch ensures that only the admin can modify the state and that the state transition is correct. The CheckProject branch uses the boolean status of a project as the outcome of the entire validation, meaning a transaction with this redeemer only succeeds if the project is marked as True.

## 4. Plastik Credit NFT Minting Policy V2 (PCMintV2.hs)

### 4.1. Scope

This is a Plutus V2 minting policy designed to mint a specific number of NFTs in a single transaction. The policy is parameterized with the minting details, including who the recipient is and how many NFTs to create. A key feature is that the designated recipient must sign the minting transaction, ensuring they consent to receiving the NFTs.

### 4.2. Implementation

This is a minting policy script, not a validator script. It governs the creation of new tokens. The policy is parameterized at compile time with an NFTDatum record, which defines the rules for a specific minting event.

### 4.3. Data Structure: Parameters (NFTDatum)

```
data NFTDatum = NFTDatum  
  
{ recipient :: PubKeyHash  
  
, amount :: Integer  
  
, nftData :: BuiltinByteString  
  
}
```

recipient: The PubKeyHash of the wallet that will receive the NFTs. This user must sign the minting transaction.

amount: The exact number of NFTs that must be minted in the transaction.

nftData: Arbitrary on-chain data associated with the NFT.

#### 4.4. Data Structure: Redeemer

The redeemer is (), the unit type, as all necessary information is provided via the script parameters and the transaction context.

Explanation: This function directly implements the policy's logic. It extracts the recipient and amount from its parameters and uses helper conditions to verify that the transaction context satisfies the two core requirements: correct mint quantity and the recipient's signature.

### 5. Plastik Credit NFT Minting Policy (PCMint.hs)

#### 5.1. Scope

This is a classic Plutus V2 one-shot minting policy. It is designed to ensure that a single, unique Non-Fungible Token (NFT) is minted. The policy's uniqueness is guaranteed by tying the minting event to the consumption of a specific, pre-determined UTxO. It also requires the signature of an authorized minter.

#### 5.2. Implementation

This minting policy is parameterized with three crucial pieces of information: an authorized minter's public key hash, a specific TxOutRef (a pointer to a UTxO), and the desired TokenName for the NFT. The script ensures that any minting transaction adheres to these parameters.

#### 5.3. Data Structure: Parameters

The policy is parameterized directly by:

pkh: The PubKeyHash of the authorized minter.

oref: The TxOutRef of the UTxO that must be spent in the minting transaction.

nftName: The TokenName of the NFT to be minted.

#### 5.4. Data Structure: Redeemer

The redeemer is (), the unit type.

Explanation: This function encapsulates the entire logic of the one-shot policy. It uses helper conditions to check each of the three critical requirements against the transaction context. By linking the mint to a consumable UTxO, it provides a robust mechanism for creating verifiably unique NFTs.

### 6. Plastik Crypto V2 Validator (PlastikCryptoV2.hs)

#### 6.1. Scope

The Plastik Crypto V2 smart contract provides a mechanism for managing NFT sales and voucher redemptions. It acts as a simple marketplace logic controller, governed by a central admin. The contract locks information about an NFT for sale and defines two distinct actions: one for redeeming a voucher and another for completing a purchase.

#### 6.2. Implementation

This is a Plutus V2 validator script parameterized by an admin PubKeyHash. The script's behavior is determined by the redeemer, which allows for two separate workflows.

RedeemVoucher: A flexible action that primarily checks for the signature of the voucher's creator.

CompleteSale: A multi-signature process that ensures the seller, buyer, and admin all authorize the completion of a sale.

#### 6.3. Data Structure: Datum

```
data SellDatum = SellDatum
  { sdSeller    :: PubKeyHash
  , sdNftPolicy :: BuiltinByteString
  , sdNftId     :: BuiltinByteString
  }
```

sdSeller: The PubKeyHash of the user selling the NFT.

sdNftPolicy: The policy ID (Currency Symbol) of the NFT being sold.

sdNftId: The token name of the NFT being sold.

#### 6.4. Data Structure: Redeemer

```
data PlastikRedeemer  
= RedeemVoucher NFTVoucher  
| CompleteSale SellRequest
```

RedeemVoucher: An action for redeeming an NFTVoucher, which contains details about the NFT and its creator.

CompleteSale: An action to finalize a sale, containing details about the token, price, seller, and buyer.

Explanation: This function routes the validation based on the chosen action. The RedeemVoucher case is a simple signature check. The CompleteSale case acts as a trustless agreement mechanism by verifying data consistency and ensuring all three participating parties (seller, buyer, admin) have signed off on the transaction.

### 7. Plastik Burner Validator (PlastikBurner.hs)

#### 7.1. Scope

The Plastik Burner contract is a stateful validator designed to manage and calculate a dynamic token burn rate. It is controlled by an owner and parameterized by a specific token's policy ID and token name. The owner can update key parameters like "plastic production" and "max burn year," which are then used to calculate the burnPerKg rate.

#### 7.2. Implementation

This Plutus V2 validator script is parameterized by the owner's PubKeyHash and the asset's identifiers. The contract's state, including the parameters for calculation, is stored in the datum. The owner can interact with the contract to update its state or trigger a read-only check of the current burn rate.

#### 7.3. Data Structure: Datum

```
data BurnerState = BurnerState
```

```

{ tokenName      :: BuiltinByteString
, policyId      :: BuiltinByteString
, totalSupply    :: Integer
, year          :: Integer
, maxBurnYear   :: Integer
, plasticProduction :: Integer
, burnPerKg     :: Integer
}

```

maxBurnYear: A configurable maximum amount of tokens to be burned in a year.

plasticProduction: A configurable figure representing plastic production.

burnPerKg: The calculated burn rate, derived from maxBurnYear and plasticProduction.

#### 7.4. Data Structure: Redeemer

```

data BurnerRedeemer
= CurrentBurnRate PubKeyHash
| SetPlasticProduction PubKeyHash Integer
| SetMaxBurnYear PubKeyHash Integer

```

CurrentBurnRate: A read-only action to check the current burn rate.

SetPlasticProduction: An owner-only action to update the plasticProduction value.

SetMaxBurnYear: An owner-only action to update the maxBurnYear value.

Explanation: The function ensures that only the owner can call the state-changing actions (SetPlasticProduction, SetMaxBurnYear). For these actions, it validates the state transition logic by comparing the expected new datum with the actual output datum. This pattern guarantees that the on-chain state evolves only according to the predefined rules.

## 8. Verified Accounts Validator (VerifiedAccounts.hs)

### 8.1. Scope

The Verified Accounts smart contract creates a flexible, on-chain identity and verification system. The system is managed by an owner who has the authority to appoint or revoke validators. These appointed validators, in turn, have the authority to update the verification status (a simple boolean) for any public key hash.

8.2. Implementation This is a Plutus V2 validator parameterized by the owner's PubKeyHash. The contract's state is maintained in the datum, which includes the list of validators and the map of verified accounts.

Two-Tiered Administration: The owner manages the set of validators. The validators manage the verification status of end-users.

Queryable Status: The contract can be used with the CheckVerification redeemer to confirm an account's status.

### 8.3. Data Structure: Datum

```
data VerifiedAccountsState = VerifiedAccountsState
    { owner      :: PubKeyHash
    , validators :: Map.Map PubKeyHash Bool
    , verifiedMap :: Map.Map PubKeyHash Bool
    }
```

owner: The PubKeyHash of the contract administrator.

validators: A map used as a set to store the PubKeyHashes of authorized validators.

verifiedMap: A map where keys are user PubKeyHashes and values are booleans indicating their verification status.

### 8.4. Data Structure: Redeemer

```
data VerifyAction
```

```
= UpdateVerification PubKeyHash Bool  
| CheckVerification PubKeyHash  
| UpdateValidator PubKeyHash Bool
```

UpdateVerification: Action to change the verification status of a target PubKeyHash.

CheckVerification: Action to check if a target PubKeyHash is verified.

UpdateValidator: Action to add or remove a validator from the authorized set.

Explanation: This function implements the two-tiered permission model. UpdateValidator is gated by an owner signature check. UpdateVerification is gated by a check against the list of authorized validators. The CheckVerification branch uses the lookup result as the validation outcome, turning the contract into a conditional spending gate based on a user's on-chain verified status.

## 9. Plastik Token On-Chain Validator (PlastikTokenOnchain.hs)

### 9.1. Scope

This smart contract implements an on-chain ledger for a fungible token. It is a stateful validator that tracks user balances, total supply, and administrative status (like whether transfers are paused). The contract is managed by an owner and can authorize lockers—special accounts that can co-sign TransferFrom transactions. This contract does not mint tokens; it only manages the ledger for tokens that already exist.

### 9.2. Implementation

This is a Plutus V2 validator script parameterized by the token's policyId, tokenName, and the owner's PubKeyHash. The entire state of the token ledger is stored in the datum of a single UTxO at the script address. All actions are performed by spending this UTxO and recreating it with an updated datum.

### 9.3. Data Structure: Datum

```
data TokenState = TokenState  
{ tokenName :: BuiltinByteString  
, paused :: Bool  
, lockers :: Map.Map PubKeyHash Bool
```

```
, balances :: Map.Map PubKeyHash Integer  
, totalSupply :: Integer  
}
```

paused: A boolean flag to enable/disable transfers.

lockers: A map of PubKeyHashes that are authorized to co-sign TransferFrom actions.

balances: A map tracking the token balance of each PubKeyHash.

#### 94. Data Structure: Redeemer

```
data TokenAction =  
  Pause | Unpause | SetLocker PubKeyHash Bool  
  | Transfer PubKeyHash Integer  
  | TransferFrom PubKeyHash PubKeyHash Integer
```

Pause, Unpause, SetLocker: Owner-only administrative actions.

Transfer: A standard token transfer from the signer to a recipient.

TransferFrom: A transfer from a senderAddr to a recipient, which must be co-signed by a locker and the sender.

Explanation: This function contains the entire validation logic. It takes the current state (ts), the desired action, and the list of transaction signers as input. For each action, it enforces the corresponding rules for pausing, signatures, and balances. For transfers, it performs the crucial check against the balances map in the provided state ts to prevent spending non-existent funds.

## **TxO Model Usage, Token Management, Security Considerations**

### **. Plastik PRG V3 On-Chain Validator**

- **TxO Model Usage:** Spends a state UTxO, which is either consumed entirely for redemptions or recreated with updated sales data by an admin.
- **Token Management:** Doesn't mint/burn assets; it tracks a pre-approved redeemer wallet and sales data in its datum.
- **Security Considerations:** Requires dual admin/redeemer signatures for withdrawals and validates that only admins can correctly modify the state.

### **. Plastik Role V2 Validator**

- **TxO Model Usage:** Manages a state UTxO that is updated when granting roles or consumed as a successful permission check for other transactions.
- **Token Management:** Manages roles, not tokens; it tracks authorized "minters" via a list of public key hashes in its datum.
- **Security Considerations:** Role granting is protected by an admin signature, while verification is a public check of the datum's list.

### **. Plastik Recovery Projects Validator**

- **TxO Model Usage:** Manages a state UTxO, which is updated by an admin for status changes or consumed as a successful on-chain verification check.
- **Token Management:** Doesn't manage tokens; tracks the boolean verification status of projects identified by public key hashes.
- **Security Considerations:** Status updates require an admin signature; verification succeeds only if a project's on-chain status is `True`.

### **. Plastik Credit NFT Minting Policy V2**

- **TxO Model Usage:** As a minting policy, it is referenced in a transaction's mint field to create new token outputs for a recipient.
- **Token Management:** Mints a specific number of NFTs, but only in a transaction that is signed by the designated recipient.
- **Security Considerations:** The core security feature is the requirement of the NFT recipient's signature, ensuring their consent.

### **. Plastik Credit NFT Minting Policy**

- **TxO Model Usage:** A minting policy that requires a specific, pre-defined UTxO to be consumed as an input for the minting transaction to be valid.
- **Token Management:** Enforces a "one-shot" minting rule, allowing only one unique NFT to ever be created by this policy.
- **Security Considerations:** Protected by an authorized minter's signature and the consumption of a unique UTxO, which prevents replays.

## . Plastik Crypto V2 Validator

- **TxO Model Usage:** Consumes a script UTxO representing an NFT sale listing to facilitate an asset swap between a buyer and seller.
- **Token Management:** Does not mint/burn tokens; its datum tracks the seller and details of the specific NFT being offered for sale.
- **Security Considerations:** Enforces a strong multi-signature scheme requiring the seller, buyer, and admin to all sign off on a completed sale.

## . Plastik Burner Validator

- **TxO Model Usage:** Manages a single state UTxO that is always consumed and recreated with an updated datum reflecting new calculation parameters.
- **Token Management:** Calculates a dynamic token burn rate based on on-chain parameters but does not execute the burn itself.
- **Security Considerations:** All actions are protected by the owner's signature, and all state transitions are strictly validated to ensure correct recalculations.

## . Verified Accounts Validator

- **TxO Model Usage:** Manages a single state UTxO which is consumed and recreated in every transaction to reflect updates to the verification or validator lists.
- **Token Management:** Manages on-chain identity, not tokens; it tracks the verification status of user accounts in its datum.
- **Security Considerations:** Uses a two-tiered system where the owner manages validators and validators manage user status, with strict state transition validation.

## . Plastik Token On-Chain Validator

- **TxO Model Usage:** Acts as a central ledger by managing a single state UTxO that must be consumed and recreated in every transaction to update balances.
- **Token Management:** Tracks the balances of a fungible token for all users in its datum but does not mint or burn the tokens itself.
- **Security Considerations:** Relies on signature checks but crucially **does not** validate the output datum, trusting the client to correctly construct the next ledger state.

## 5. Data Flow & Transaction Lifecycle

### Data Flow Diagram



- Non-Fungible Asset (NFT) Framework:** The platform includes multiple **minting policies** for creating diverse types of NFTs. These range from classic "one-shot" policies that guarantee uniqueness by consuming a specific UTxO, to more flexible policies that allow for batch minting with recipient consent. This framework is complemented by validator scripts designed to handle marketplace sales and voucher redemptions.
- Identity Layer:** A sophisticated, two-tiered model underpins the entire system. An **owner** role has ultimate administrative control, with the ability to appoint **validators** or **minters**. These designated roles are then granted specific on-chain permissions, such as verifying user accounts and managing operational lists. This creates a clear and secure hierarchy for system management.
- Application-Specific Logic:** The architecture is designed to support specialized business logic through dedicated validator scripts. Contracts for managing **Plastic Recovery Projects** and calculating a dynamic **token burn rate** showcase how the platform's core components can be leveraged to build transparent, auditable, and automated workflows specific to the project's goals.

Collectively, these smart contracts form a cohesive decentralized application. By leveraging stateful UTxOs for on-chain ledgers and parameterized scripts for access control, the system minimizes trust assumptions and ensures that all critical operations are executed and verified on the Cardano blockchain.

## 6. Off-chain Components

- Accessing external data (e.g., calling a weather API or a carbon registry).
  - Performing complex, rapid calculations that don't need to be on-chain.
  - Storing large amounts of user data (like profiles or historical reports).
  - Providing a fast, responsive user interface.
  - Backend:-
- Business Logic:** It handles all the complex logic that doesn't need to be run by every node on the blockchain. For your carbon credit platform, this would include user registration, project submissions, and calculating metrics.

**Database Management:** It maintains a traditional database (MySQL) to store data that is inefficient or too expensive to keep on-chain. This includes:

- User profiles (email, name, company info).
- Project details (descriptions, images, documentation links).
- A cached/indexed copy of on-chain events for fast querying.

**Transaction Building:** Constructing Cardano transactions can be complex. The backend can do the heavy lifting of gathering the right UTxOs, calculating fees, and building the transaction structure before sending it to the user's wallet for the final signature.

**Serving the Frontend:** It hosts the files for your user-facing dashboard.

Integration with cardano node-----

Blockfrost: A service that runs the complex Cardano infrastructure for you and provides a simple REST API. You just make HTTPS requests to their endpoints.

Oracle- A smart contract cannot access data outside of the blockchain. An **oracle** is a trusted entity that bridges this gap by bringing real-world, off-chain data *onto* the blockchain in a way the smart contract can use.

## 7. Sandbox/Testnet Results

The Test Report will be shared separately.

## 8. Tools and Environments Used

- The following commands:
  - cabal clean //Removes previous build artifacts to ensure a clean slate and avoid conflicts.
  - cabal update //Refreshes the local package repository index so that dependencies are up-to-date.
  - cabal build //Compiles the smart contract code into an executable form.
  - cabal repl //Launches an interactive environment for testing your smart contract functions.
- Generate the CBOR Hex File
- GHC – 8.10.7
- Cabal – 3.8.1.0
- Haskell and Plutus
- Cardano Node (Conway era) (10.4.1)
- Cardano addresses 4.0.0
- Deploy the Smart Contract
  - Blockchain: Cardano Blockchain that supports Plutus.
  - Deployment: Deploy the smart contract to the blockchain.
- Smart Contract Test
  - cabal clean //Removes previous build artifacts to ensure a clean slate and avoid conflicts.
  - cabal update //Refreshes the local package repository index so that dependencies are up-to-date.
  - cabal build //Compiles the smart contract code into an executable form.
  - cabal repl //Launches an interactive environment for testing your smart contract functions.

## 9. Remaining Considerations / Next Steps

Here are the remaining considerations and next steps for the project

### Pending Architecture Changes

We will also explore upgrading single-admin roles to a multi-signature or DAO-based model to enhance decentralization and security.

### Performance Improvements

For production deployment, all debugging functions will be removed to reduce script size and execution costs. Furthermore, for more efficient on-chain operations to reduce the fees.

### Security Audit Steps

A comprehensive security audit is the next priority. This will involve a review of all validator logic, formal verification where applicable, and an economic analysis to identify potential exploit vectors. Special focus will be placed on the client-side state validation in the token ledger contract.