creating an advance level smart contract in aiken-lang where have rewards minting list delist buy sell explore everything with using aiken-lang new features and functions

Creating an advanced smart contract using Aiken-lang's new features and functionality requires a deep understanding of the language's capabilities. We'll focus on building a marketplace that includes features such as rewards minting, listing, delisting, buying, selling, and exploring assets. This contract will use new Aiken functions and modules that provide efficient validation and interaction with on-chain data.

step by step, using Aiken-lang's modern features:

1. Contract Overview

This contract provides:

- **Listing and Delisting**: Users can list or delist assets on the marketplace.
- **Buying and Selling**: Users can buy listed assets using ADA.
- **Rewards Minting**: Users receive rewards after specific conditions, like making multiple purchases.
- **Exploration**: Users can view and explore all active listings.

We will structure this contract around a central marketplace state that holds the data for all listings and rewards.

2. Data Types

We'll define data types for assets, listings, and marketplace state.

aiken

```
type Asset {
  id: ByteArray        -- Unique asset identifier
  owner: PubKeyHash    -- Owner's public key
  price: Int           -- Asset price in lovelace
}

type Listing {
  asset: Asset         -- The asset being listed
  active: Bool         -- Whether the listing is active
}

type MarketplaceState {
  listings: Map ByteArray Listing   -- Mapping from asset IDs to listings
  rewards: Map PubKeyHash Int       -- Mapping from user PKHs to their rewards
}
```

- **Asset**: A digital asset with an ID, owner, and price.
- **Listing**: Represents the state of an asset in the marketplace (listed or delisted).
- **MarketplaceState**: Tracks all listings and user rewards.

3. Contract Logic

We'll write contract functions to handle listing, delisting, buying, selling, and reward minting. Each function will follow a specific set of rules to validate the transaction and update the state.

3.1. Listing an Asset

This function allows a user to list an asset on the marketplace.

aiken

```
validator list_asset(datum: PubKeyHash, redeemer: Asset, context: ScriptContext) -> Bool {
  let tx_info = context.tx_info
  let asset_id = redeemer.id
  let new_listing = Listing {
    asset: redeemer,
    active: true
  }
```

```
    let listings = find_in_script_datum(context.input_datum)
    let new_listings = Map.insert(asset_id, new_listing, listings)

    assert.contains(tx_info.signatories, redeemer.owner) -- Ensure the owner is the one listing the
asset

    ScriptContext.output_datum_is(context.output_datum, new_listings)
}
```

- **Logic**: The owner of the asset lists it by passing the asset's data as the redeemer. The contract checks that the owner is signing the transaction and adds the listing to the marketplace.

3.2. Delisting an Asset

Users can delist their assets from the marketplace.

```
aiken
validator delist_asset(datum: PubKeyHash, redeemer: ByteArray, context: ScriptContext) ->
Bool {
    let tx_info = context.tx_info
    let asset_id = redeemer

    let listings = find_in_script_datum(context.input_datum)
    let listing = Map.get(asset_id, listings)

    assert.some(listing, "Listing not found")
    assert.equal(listing.active, true) -- Asset must be active to delist
    assert.contains(tx_info.signatories, listing.asset.owner) -- Owner must be the one delisting

    let updated_listings = Map.insert(asset_id, {listing with active = false}, listings)
    ScriptContext.output_datum_is(context.output_datum, updated_listings)
}
```

Logic: The asset owner provides the asset ID in the redeemer, and the contract verifies that the listing is active and the owner is signing the transaction before deactivating the listing.

3.3. Buying an Asset

When a user buys an asset, the contract will transfer ADA and update the marketplace state.

aiken

```
validator buy_asset(datum: PubKeyHash, redeemer: ByteArray, context: ScriptContext)
-> Bool {
  let tx_info = context.tx_info
  let asset_id = redeemer

  let listings = find_in_script_datum(context.input_datum)
  let listing = Map.get(asset_id, listings)

  assert.some(listing, "Asset not listed")
  assert.equal(listing.active, true) -- Asset must be listed
  assert.contains(tx_info.signatories, datum) -- Buyer must sign

  -- Ensure the buyer has enough ADA to buy
  let payment_valid = ScriptContext.payment_value_is(context, listing.asset.price)
  assert.equal(payment_valid, true)

  -- Update state: mark asset as sold and transfer ADA
  let new_listing = { listing with active = false }
  let updated_listings = Map.insert(asset_id, new_listing, listings)

  ScriptContext.output_datum_is(context.output_datum, updated_listings)
}
```

- **Logic: The buyer submits the asset ID and provides sufficient ADA. The contract ensures the asset is still listed and marks it as sold.**

3.4. Minting Rewards

Rewards are minted after specific actions, such as when a user makes a certain number of purchases.

```
aiken
validator mint_rewards(datum: PubKeyHash, redeemer: Int, context:
ScriptContext) -> Bool {
  let tx_info = context.tx_info
  let buyer = datum
  let reward_threshold = 5

  let rewards = find_in_script_datum(context.input_datum)
  let current_reward = Map.get(buyer, rewards) | 0

  let new_reward = current_reward + redeemer
```

```
    -- Mint reward if the user reaches the threshold
    if new_reward >= reward_threshold {
        let rewards_with_mint = Map.insert(buyer, 0, rewards) -- Reset reward after
mint
        ScriptContext.mint_token(buyer, reward_threshold)
    } else {
        let updated_rewards = Map.insert(buyer, new_reward, rewards)
        ScriptContext.output_datum_is(context.output_datum, updated_rewards)
    }
}
```

- Logic: Each time a user buys an asset, their reward count is updated. When they reach the reward threshold (e.g., 5 purchases), the contract mints a reward for them and resets their counter.

3.5. Exploring Active Listings

This function returns a list of active listings.

```
validator explore_marketplace(datum: PubKeyHash, redeemer: Unit, context:
ScriptContext) -> Bool {
    let listings = find_in_script_datum(context.input_datum)

    -- Filter only active listings
    let active_listings = Map.filter((listing) -> listing.active, listings)

    -- The output must reflect active listings
    ScriptContext.output_datum_is(context.output_datum, active_listings)
}
```

Logic: The contract returns all active listings. Users can explore the marketplace by calling this function, which filters for assets that are still listed.