ERC-20 Token Contract Documentation

The ERC-20 Token Contract sets up a scenario to manage token transactions between users, ensuring that each transaction is valid and updating balances accordingly. The contract uses the Equivalence Principle to maintain consistency and accuracy.

```python
import json
from backend.node.genvm.icontract import IContract
from backend.node.genvm.equivalence_principle import EquivalencePrinciple


class LlmErc20(IContract):
    def __init__(self, total_supply: int) -> None:
        self.balances = {}
        self.balances[contract_runner["from_address"]] = total_supply

    async def transfer(self, amount: int, to_address: str) -> None:
        prompt = f"""
You keep track of transactions between users and their balance in coins.
The current balance for all users in JSON format is:
{json.dumps(self.balances)}
The transaction to compute is: {{
sender: "{contract_runner["from_address"]}",
recipient: "{to_address}",
amount: {amount},
}}

For every transaction, validate that the user sending the Coins has
enough balance. If any transaction is invalid, it shouldn't be processed.
Update the balances based on the valid transactions only.
Given the current balance in JSON format and the transaction provided,
please provide the result of your calculation with the following format:
{{
transaction_success: bool,          // Whether the transaction was successful
transaction_error: str,             // Empty if transaction is successful
updated_balances: object<str, int>  // Updated balances after the transaction
}}
It is mandatory that you respond with only the JSON output object, nothing
else whatsoever. Don't include any other words or characters, your
output must be only the JSON object"""

        final_result = {}
        async with EquivalencePrinciple(
            result=final_result,
            principle="""The new_balance of the sender should have decreased
            in the amount sent and the new_balance of the receiver should have
            increased by the amount sent. Also, the total sum of all balances
            should have remain the same before and after the transaction""",
            comparative=True,
        ) as eq:
            result = await eq.call_llm(prompt)
            result_clean = result.replace("True", "true").replace("False", "false")
            eq.set(result_clean)

        result_json = json.loads(final_result["output"])
        self.balances = result_json["updated_balances"]

    def get_balances(self) -> dict[str, int]:
        return self.balances

    def get_balance_of(self, address: str) -> int:
        return self.balances.get(address, 0)
```

Code Explanation

- **Initialization**: The `LlmErc20` class initializes the contract with a total supply of tokens assigned to the contract creator's address. `[contract_runner["from_address"]]` represents the address that called the contract function, helping to track and validate transactions.
- **Prompts**: Next, we construct prompts to ensure that the LLM can accurately validate and process transactions based on the provided user balances and transaction details (sender, recipient, and amount). This prompt is sent to the language model (LLM) for processing and validation.
- **EquivalencePrinciple**: The Equivalence Principle validates that the sender's balance decreases by the amount sent, the recipient's balance increases by the same amount, and the total sum of all balances remains unchanged.
- **JSON Response**: The LLM processes the prompt and returns a JSON response indicating the transaction status and updated balances.
- **Balance Update**: The contract updates the balances based on the LLM response.
- **Balance Retrieval Methods**:
  - `get_balances()`: Returns the balances of all users.

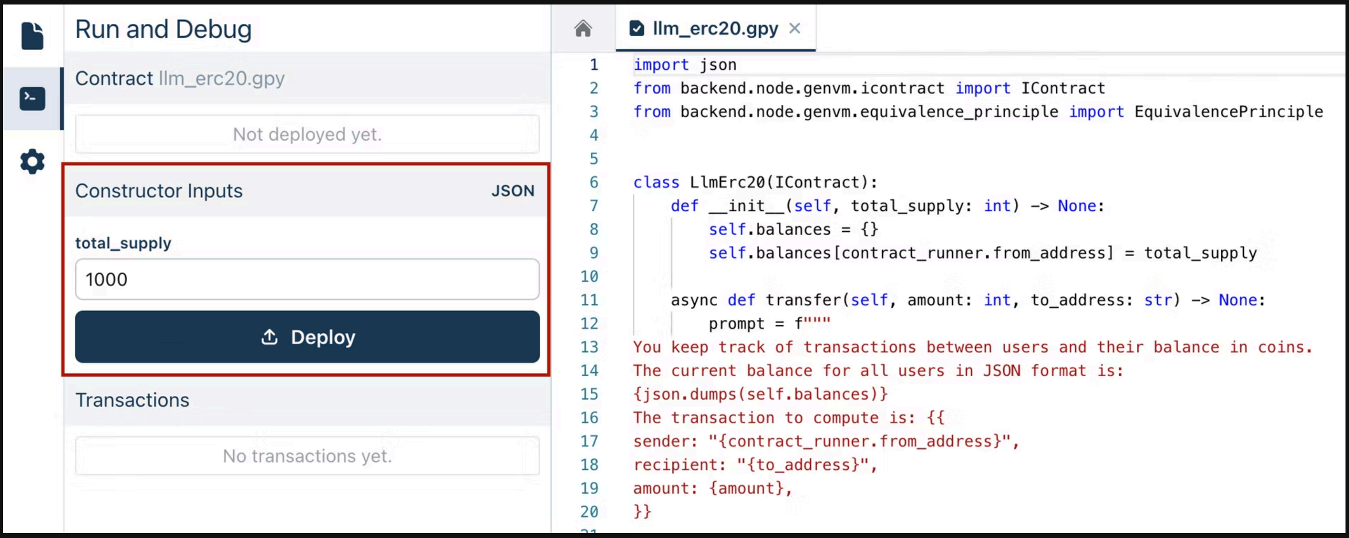
- `get_balance_of(address)`: Returns the balance of a specific user given their address.

👾 You can view this code on our GitHub.

Deploying the Contract

To deploy the ERC-20 Token contract, you need to initialize the contract state correctly. This setup will determine how the contract manages token transactions.

1. **Set Total Supply**: Provide the total supply of tokens. The `total_supply` constructor parameter is detected from the code. For example, you might set `total_supply` to 1000.
2. **Deploy the Contract**: Once the total supply is set, deploy the contract to make it ready to interact and process token transactions.

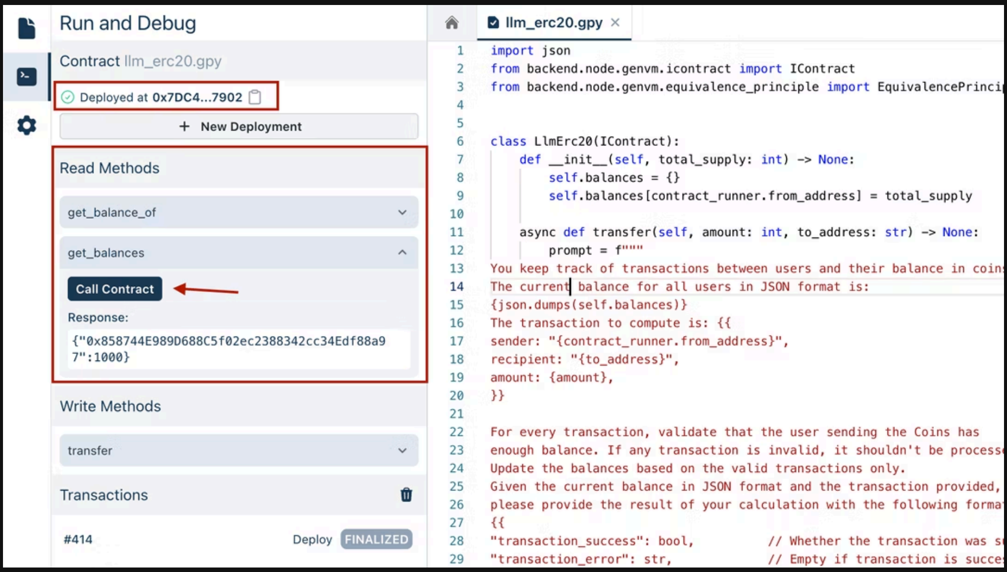


total_supply: 1000

```
import json
from backend.node.genvm.icontract import IContract
from backend.node.genvm.equivalence_principle import EquivalencePrinciple


class LlmErc20(IContract):
    def __init__(self, total_supply: int) -> None:
        self.balances = {}
        self.balances[contract_runner.from_address] = total_supply

    async def transfer(self, amount: int, to_address: str) -> None:
        prompt = f"""
You keep track of transactions between users and their balance in coins.
The current balance for all users in JSON format is:
{json.dumps(self.balances)}
The transaction to compute is: {{
sender: "{contract_runner.from_address}",
recipient: "{to_address}",
amount: {amount},
}}
```

Checking the Contract State

After deploying the contract, you can check its state in the **Current Intelligent Contract State** section. This section displays the contract address and the current account. Use the `get_balances()` function to see the balances of all users and the `get_balance_of(address)` function to see the balance of a specific user.

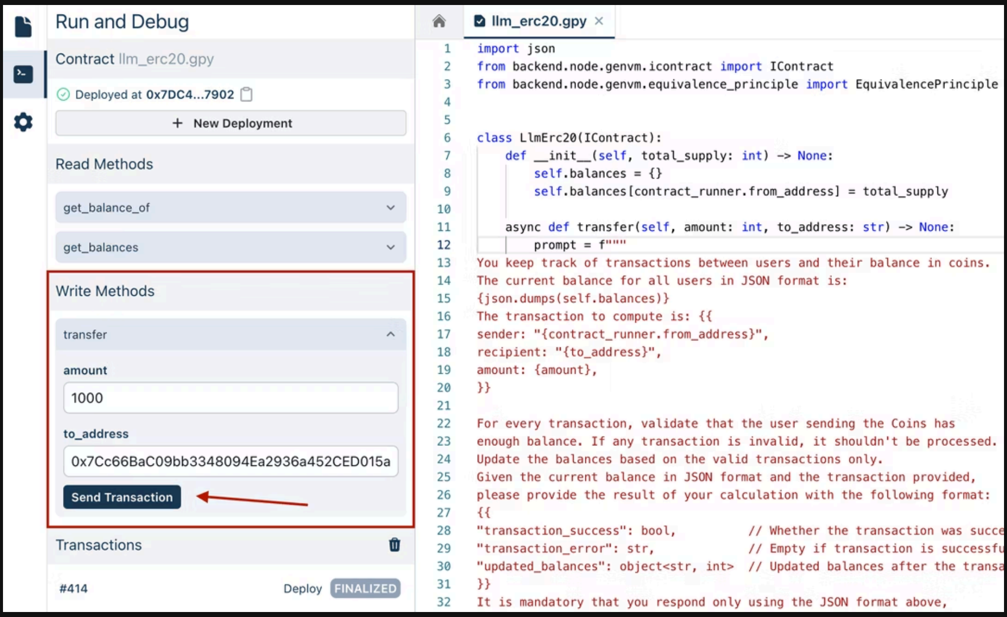


```
{"0x858744E989D688C5f02ec2388342cc34Edf88a9
7":1000}
```

```
import json
from backend.node.genvm.icontract import IContract
from backend.node.genvm.equivalence_principle import EquivalencePrinci


class LlmErc20(IContract):
    def __init__(self, total_supply: int) -> None:
        self.balances = {}
        self.balances[contract_runner.from_address] = total_supply

    async def transfer(self, amount: int, to_address: str) -> None:
        prompt = f"""
You keep track of transactions between users and their balance in coin
The current balance for all users in JSON format is:
{json.dumps(self.balances)}
The transaction to compute is: {{
sender: "{contract_runner.from_address}",
recipient: "{to_address}",
amount: {amount},
}}

For every transaction, validate that the user sending the Coins has
enough balance. If any transaction is invalid, it shouldn't be processe
Update the balances based on the valid transactions only.
Given the current balance in JSON format and the transaction provided,
please provide the result of your calculation with the following forma
{{
"transaction_success": bool,          // Whether the transaction was s
"transaction_error": str,             // Empty if transaction is succe
```

Executing Transactions

To interact with the deployed contract, go to the **Execute Transactions** section. Here, you can call the `transfer` method to process a token transfer. This triggers the contract's logic to validate the transaction and update balances based on the Equivalence Principle criteria defined.



amount: 1000

to_address: 0x7Cc66BaC09bb3348094Ea2936a452CED015a

```
import json
from backend.node.genvm.icontract import IContract
from backend.node.genvm.equivalence_principle import EquivalencePrinciple


class LlmErc20(IContract):
    def __init__(self, total_supply: int) -> None:
        self.balances = {}
        self.balances[contract_runner.from_address] = total_supply

    async def transfer(self, amount: int, to_address: str) -> None:
        prompt = f"""
You keep track of transactions between users and their balance in coins.
The current balance for all users in JSON format is:
{json.dumps(self.balances)}
The transaction to compute is: {{
sender: "{contract_runner.from_address}",
recipient: "{to_address}",
amount: {amount},
}}

For every transaction, validate that the user sending the Coins has
enough balance. If any transaction is invalid, it shouldn't be processed.
Update the balances based on the valid transactions only.
Given the current balance in JSON format and the transaction provided,
please provide the result of your calculation with the following format:
{{
"transaction_success": bool,          // Whether the transaction was succe
"transaction_error": str,             // Empty if transaction is successfu
"updated_balances": object<str, int>  // Updated balances after the transa
}}
It is mandatory that you respond only using the JSON format above,
```

Analyzing the Contract's Decisions

When the `transfer` method is executed:

- The LLM processes the transaction details.
- It validates the transaction according to the Equivalence Principle defined in the code.
- It returns a JSON response that includes the transaction status and updated balances.

Handling Different Scenarios

- **Valid Transactions**: If the sender has enough balance, the transaction is successful. The sender's balance decreases by the amount sent, and the recipient's balance increases by the same amount. The total sum of all balances remains unchanged.
- **Invalid Transactions**: If the sender does not have enough balance, the transaction fails. The balances remain unchanged, and the JSON response includes an error message.

You can view the logs to see detailed information about the contract interaction.