BOSON PROTOCOL

Infrastructure Engineer Technical Interview - Problem Statement

Context

Boson Protocol is advancing the world of commerce by enabling enterprises, organisations, and customers to bridge the divide between digital decentralized technologies and the transfer and trade of physical goods.

Our vision is for Boson to become the basic plumbing for dCommerce and its data on the emerging decentralized web, where the value captured is distributed equitably between token holders and protected from capture by a single centralized entity. Read more about us here: https://bosonprotocol.io/

Guidelines

- The following exercise contains some example invocations, however the solution should be extendable to support similar invocations.
- You can solve the problem in the language of your choice, although we might not be familiar with it, so be prepared to explain it! Bear in mind that we use Ruby for all of our automation code.
- Please submit the solution as an email attachment (.zip).
- What we will look for:
    - a good breakdown of the core domain,
    - a simple interface, we don't expect UIs, web APIs or database integration
    - a flexible solution, that can be extended in various different ways,
    - a well tested solution,
    - a simple solution, however a library shouldn't solve the problem for you
    - clear instructions on how to run the solution and tests,
    - a brief explanation of your design and assumptions,
- We want our hiring practices to be fair. To ensure this, please do not publically share the problem or solution.

Problem

The Boson Protocol core smart contracts govern the movement of funds into and out of escrow for all commerce transactions based on the protocol. As a result, they will be responsible for potentially huge amounts of money at any given time. To minimise the risk of financial loss, a number of administrative controls have been built into the contracts. One such administrative control is being able to completely pause the contracts, effectively halting all activity on the protocol.

The core protocol team would like to be able to make administrative changes through a small CLI and they have asked you to help build it. Your mission is to:

- Deploy a local testing node
    - For this you can use Docker or a locally running process
    - You may also use a tool like `ganache-cli` to simplify running the node
- Deploy the contract to the testing node
- Build a simple CLI that operates against this testing node and provides pause / unpause behaviour

A cut-down version of the contract that you will be interacting with is included in the appendix.

Note that if we were really building this tool, we would require multiple signatories to trigger these administrative controls, using something like Gnosis Safe. However, for the purposes of this exercise, consider that out of scope.

Example executions:

Below, `owner-mnemonic` is the mnemonic used to unlock the wallet of the account that deployed the contract and `contract-address` is the address at which the contract was deployed.

```
$ boson \
>   --node=127.0.0.1:8545 \
>   --mnemonic=<owner-mnemonic> \
>   --address=<contract-address> \
>   status
Unpaused

$ boson \
>   --node=127.0.0.1:8545 \
>   --mnemonic=<owner-mnemonic> \
>   --address=<contract-address> \
>   pause
Paused

$ boson \
>   --node=127.0.0.1:8545 \
>   --mnemonic=<owner-mnemonic> \
>   --address=<contract-address> \
>   status
Paused

$ boson \
>   --node=127.0.0.1:8545 \
>   --mnemonic=<owner-mnemonic> \
>   --address=<contract-address> \
>   unpause
Unpaused

$ boson \
>   --node=127.0.0.1:8545 \
>   --mnemonic=<owner-mnemonic> \
>   --address=<contract-address> \
>   status
Unpaused
```

Appendix

```
// SPDX-License-Identifier: LGPL-3.0-or-later
pragma solidity 0.7.1;

/**
 * @title Contract for interacting with Boson Protocol from the user's
 *        perspective.
 */
contract BosonProtocol {
    /**
     * @notice Emitted when the pause is triggered by `account`.
     */
    event Paused(address account);

    /**
     * @notice Emitted when the pause is lifted by `account`.
     */
    event Unpaused(address account);

    address private _owner;
    bool private _paused;

    /**
     * @notice Initializes the contract in unpaused state.
     */
    constructor () {
        _owner = msg.sender;
        _paused = false;
    }

    /**
     * @notice Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }


    /**
     * @notice Returns true if the contract is paused, and false otherwise.
     */
    function paused() public view returns (bool) {
        return _paused;
    }

    /**
     * @notice Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(owner() == msg.sender, "Caller is not the owner");
        _;
    }

    /**
     * @notice Modifier to make a function callable only when the contract is not
     *         paused.
     *
     * Requirements:
     *
     * - The contract must not be paused.
     */
    modifier whenNotPaused() {
        require(!paused(), "Paused");
        _;
    }

    /**
     * @notice Modifier to make a function callable only when the contract is
     *         paused.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    modifier whenPaused() {
        require(paused(), "Not paused");
        _;
    }

    /**
     * @notice Triggers stopped state.
     *
     * Requirements:
     *
     * - The contract must not be paused.
     */
    function pause() external whenNotPaused onlyOwner {
        _paused = true;
        emit Paused(msg.sender);
    }

    /**
     * @notice Returns to normal state.
     *
     * Requirements:
     *
     * - The contract must be paused.
     */
    function unpause() external whenPaused onlyOwner {
        _paused = false;
        emit Unpaused(msg.sender);
    }
}
```