Smart Contract Deployment API with Contract Database and Interaction

Introduction

This take-home assignment assesses your backend development skills in the context of blockchain technology. You will build an API that manages the deployment of smart contracts using Hardhat, stores deployment information in a database, and provides endpoints to interact with the deployed contract. While a basic frontend is required for testing purposes, the primary focus of this assignment is the design and implementation of the backend API.

Problem Statement

Create a backend API that provides the following functionality:

- **Start Hardhat Node:** An API endpoint should initiate a local Hardhat node in the background.
- **Deploy Smart Contract:** An API endpoint should deploy the provided Counter.sol Solidity smart contract to the running Hardhat node.
- **Store Deployment Information:** Upon successful deployment, the API should store the following information in a database:
    - Contract Address
    - Deployment Timestamp
    - (Optional but recommended) Transaction Hash
- **Retrieve Deployed Contracts:** An API endpoint should retrieve a list of all deployed contracts from the database, including the information stored above.
- **Increment Counter:** An API endpoint should call the increment() method of a specified deployed contract.
- **Get Counter Value:** An API endpoint should read and return the current count value of a specified deployed contract.

Frontend (Minimal Requirement)

You are required to create a *minimal* frontend to interact with your API. This frontend should have at least **four buttons** and **a display area**:

- **"Start Hardhat Node"**: Calls the appropriate API endpoint.
- **"Deploy Contract"**: Calls the appropriate API endpoint. After deployment, display the contract address.
- **"Increment Counter"**: Calls the API endpoint to increment the counter of the *most recently deployed* contract.
- **"Get Counter Value"**: Calls the API endpoint to retrieve and display the count value of the *most recently deployed* contract in a designated display area.

The frontend is primarily a tool for testing your backend API and will be weighted less heavily in the evaluation.

Technical Requirements

- **Backend (Required):** Use a backend framework (e.g., Node.js with Express, Python with Flask/Django, etc.).
- **Database (Required):** Use a database (e.g., PostgreSQL, MySQL, MongoDB, SQLite) to store information about deployed contracts. Justify your database choice in the README.
- **Smart Contract:** Use the provided Counter.sol smart contract (see below).
- **Hardhat:** Use Hardhat as the development environment.
- **Web3 Library:** Use a Web3 library (e.g., ethers.js, web3.py) in your backend to interact with the deployed smart contract.
- **Communication:** The backend should interact with the Hardhat node, the smart contract (via the Web3 library), and the database. The frontend should communicate with the backend API.

Provided Resources

- **Counter.sol**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Counter {
    uint256 public count;

    constructor() {
        count = 0;
    }

    function increment() public {
        count++;
    }
}
```

Deliverables

- **Source Code:** A well-organized, commented, and easily understandable codebase. Focus on clean backend code.
- **README:** A comprehensive README file containing:
    - Clear instructions for setting up and running the application (including any necessary dependencies for both the backend and minimal frontend).
    - Complete API documentation, including endpoints, request/response formats, and example usage.
    - Database schema description and justification for the database choice.
    - Explanation of your backend design choices, including how you manage the Hardhat node process, interact with the database, and interact with the smart contract.
    - Any known limitations or potential improvements.

Evaluation Criteria

- **Backend Functionality (Highly Weighted):** Does the API meet all backend requirements (starting Hardhat, deploying contract, storing and retrieving contract information, incrementing counter, getting counter value)?
- **Backend Code Quality (Highly Weighted):** Is the backend code clean, well-structured, efficient, and easy to understand? Are appropriate design patterns used?
- **Database Interaction (Highly Weighted):** Are database interactions efficient, secure, and correctly implemented? Is the database schema well-designed? Justification for database choice.
- **Smart Contract Interaction (Highly Weighted):** Is the interaction with the smart contract (incrementing and getting the value) correctly implemented using a Web3 library?
- **API Design (Moderately Weighted):** Is the API well-designed, RESTful (if applicable), and easy to use?
- **Hardhat Integration (Moderately Weighted):** Is Hardhat correctly integrated for compiling and deploying the contract?
- **Error Handling (Moderately Weighted):** Does the API handle errors gracefully and provide informative error messages?
- **Security Considerations (Moderately Weighted):** Are security best practices followed?
- **Frontend Implementation (Lightly Weighted):** Is the minimal frontend functional and easy to use for testing the API? Detailed frontend design is not expected.
- **Documentation (Moderately Weighted):** Is the README clear, concise, and comprehensive?