Blockchain & Solidity Lab2 – Voting dApp Development

S2BC

Lab 2: Testing Ethereum Smart Contracts with Hardhat

- BUILD / **TEST** / INTEGRATE / RUN

Objective: In this lab, we will focus on testing Ethereum Smart Contracts using Hardhat. Testing is a crucial step in the development process to ensure the reliability and security of your smart contracts.

Table of Contents

1. Introduction to Testing Ethereum Smart Contracts

In this section, we'll explore the importance of testing smart contracts and how it ensures the integrity of the blockchain application. Testing helps identify and fix potential vulnerabilities in your code before deployment.

1.1. The Importance of the Testing Object in Solidity

The testing object in Solidity serves as a fundamental element in crafting exhaustive test cases. This pivotal component enables the simulation of diverse scenarios and interactions with your smart contracts, ensuring their seamless functionality in accordance with your design intentions.

1.2. Running the Tests

To run tests using Hardhat, follow these steps:

1. Find the hardhat/tests directory and write test files Voting.test.js
2. Use the Hardhat command-line interface (CLI) to execute the tests.
3. Review the output for any failed tests and debug accordingly.

1.3. Best Practices for Smart Contract Testing

Writing effective test cases is crucial for contract security. Here are some best practices to consider:

- Use descriptive test case names to clearly indicate the purpose of each test.
- Write assertions to validate contract behavior. This ensures that contracts function as expected.
- Test edge cases and potential failure scenarios to cover all possible outcomes.

1.4. Benefits of Testing Ethereum Smart Contracts with Hardhat

Testing Ethereum smart contracts using Hardhat provides several advantages that contribute to the reliability and security of your blockchain application. Here are the key benefits:

1.5. Time Savings on Testing

Hardhat creates a virtual blockchain environment for deploying and testing smart contracts. This allows developers to test their contract functions without interacting with the main Ethereum network. By leveraging this local testing environment, you save valuable time compared to deploying and testing on the live network. Fast iterations in a controlled environment enhance development efficiency.

1.6. Early Bug Detection

Testing smart contracts with Hardhat enables developers to identify and fix potential vulnerabilities in the code before deployment to the mainnet. By simulating various scenarios and interactions through comprehensive test cases, you can catch bugs and issues early in the development process. This proactive approach reduces the risk of deploying faulty contracts, enhancing the overall security of your blockchain application.

1.7. Improved Code Quality

Writing test cases encourages developers to follow best coding practices and design patterns. As you create tests to validate different aspects of your smart contract functionality, you naturally structure your code in a modular and organized manner. This not only makes the codebase more maintainable but also enhances collaboration among team members.

1.8. Documentation Through Tests

Test cases serve as a form of documentation for your smart contracts. By examining the test suite, developers can quickly understand the expected behavior of each function and the contract as a whole. This documentation becomes especially valuable when onboarding new team members or revisiting the code after a period of time.

1.9. Regression Testing

As your smart contract evolves with new features or optimizations, running the existing test suite ensures that the changes do not introduce regressions. Regression testing is crucial for maintaining the integrity of the codebase over time. Hardhat simplifies this process by providing a reliable testing framework that can be easily integrated into your development workflow.

Conclusion

Incorporating comprehensive testing practices with Hardhat is not just a best practice; it's a fundamental step toward building secure and reliable Ethereum smart contracts. By investing time in testing during the development phase, you mitigate risks, improve code quality, and contribute to the overall success of your blockchain project.

---

2. Writing Tests for the "Voting" Smart Contract

2.1. **Create a Test File:**

- Create a new file named `Voting.test.js` in your hardhat/test folder.

2.2. **Import Dependencies:**

- Import the necessary dependencies, including the testing library (chai) and ethers.

```
const { expect } = require("chai");
```

2.3. **Setup Test Environment:**

- Create a describe block for the "Voting Contract" test suite. Inside, declare variables for the Voting contract, deployed instance, owner, and two additional addresses.

```
describe("Voting Contract", function () {
    let Voting;  // Declare variable for the Voting contract
    let voting;  // Declare variable for the deployed instance of the
Voting contract
    let owner;   // Declare variable for the owner of the contract
    let addr1;   // Declare variable for address 1
    let addr2;   // Declare variable for address 2
```

2.4. **Deploy a Fresh Instance Before Each Test:**

- Use the `beforeEach` hook to deploy a fresh instance of the Voting contract before each test.

```
beforeEach(async function () {
    [owner, addr1, addr2] = await ethers.getSigners();

    const VotingFactory = await ethers.getContractFactory("Voting");
    voting = await VotingFactory.deploy();
    Voting = await VotingFactory.connect(owner);
});
```

2.5. **Write Test Cases:**

- Write test cases to check the correct owner after deployment, starting an election with the correct parameters, and handling voter registration and voting check.

```
it("should have the correct owner after deployment", async function () {
    expect(await voting.owner()).to.equal(owner.address);
});

it("should start an election with the correct parameters", async function
() {
    // Test logic for starting an election
    // Assertions related to election start
});

it("should handle voter registration and voting", async function () {
    // Test logic for voter registration and voting
    // Assertions related to the voting process
});
```

2.6. **Example Test Logic for "Start an Election":**

- Within the second test case, write logic to start an election with specific parameters and make assertions related to the election start.

```
it("should start an election with the correct parameters", async function
() {
    const electionTitle = "Test Election";
    const candidates = ["Candidate1", "Candidate2"];
    const votingDuration = 10; // in minutes

    await voting.startElection(electionTitle, candidates, votingDuration);

    expect(await voting.electionStarted()).to.be.true;
    // Additional assertions related to the election start
    // ...
});
```

2.7. **Closing the Describe Block:**

- Close the describe block.

```
});
```

3. Voting.test.js

```

```

```
const { expect } = require("chai");

// Describe block for the Voting Contract test suite
describe("Voting Contract", function () {
    let Voting;  // Declare variable for the Voting contract
    let voting;  // Declare variable for the deployed instance of the
Voting contract
    let owner;   // Declare variable for the owner of the contract
    let addr1;   // Declare variable for address 1
    let addr2;   // Declare variable for address 2

    // Before each test case, deploy a fresh instance of the Voting
contract
    beforeEach(async function () {
        // Get signers (addresses) from ethers
        [owner, addr1, addr2] = await ethers.getSigners();

        // Deploy the Voting contract using the factory
        const VotingFactory = await ethers.getContractFactory("Voting");
        voting = await VotingFactory.deploy();

        // Connect the contract factory to the owner
        Voting = await VotingFactory.connect(owner);
    });

    // Test case 1: Should check if the correct owner is set after
deployment
    it("should have the correct owner after deployment", async function ()
{
        // Assert that the owner of the contract is equal to the expected
owner's address
        expect(await voting.owner()).to.equal(owner.address);
    });

    // Test case 2: Should start an election with the correct parameters
    it("should start an election with the correct parameters", async
function () {
        const electionTitle = "Test Election";
        const candidates = ["Candidate1", "Candidate2"];
        const votingDuration = 10; // in minutes

        // Start an election with the specified parameters
        await voting.startElection(electionTitle, candidates,
votingDuration);

        // Additional assertions related to the election start
        expect(await voting.electionStarted()).to.be.true;
        expect(await voting.votingStartTimeStamp()).to.not.equal(0);
        expect(await voting.votingEndTimeStamp()).to.not.equal(0);
        expect(await voting.electionTitle()).to.equal(electionTitle);
    });

    // Test case 3: Should handle voter registration and voting
```

```
    it("should handle voter registration and voting", async function () {

       // Start an election before registering voters and casting votes
        const electionTitle = "Test Election";
        const candidates = ["Candidate1", "Candidate2"];
        const votingDuration = 10; // in minutes
        await voting.startElection(electionTitle, candidates,
votingDuration);

        // Register voters for this specific test scenario
        await voting.registerVoter(addr1.address);
        await voting.registerVoter(addr2.address);
        await voting.registerVoter(owner.address);

        // Cast votes for Candidate with ID 0
        await voting.connect(addr1).voteTo(0);
        await voting.connect(addr2).voteTo(0);
        await voting.connect(owner).voteTo(0);

        // Additional assertions related to the voting process
        const voteCountCandidate0 = (await voting.retrieveVotes())
[0].numberOfVotes;

        // Assert various conditions for the voting process, here all
voters voted for candidate ID 0, so he should get 3 votes.
        expect(voteCountCandidate0).to.equal(3); // Assuming three voters
    });

});
```

4. Running the test

1. Make sure you are located into "voting-dapp-2023/hardhat", directory where your `hardhat.config.js` file is located.
2. Run the following command to execute the tests:

```
npx hardhat test
```

Hardhat will automatically detect and run all the test files in your `test` directory. It will then display the test results, indicating whether each test case passed or failed.

The output should look like this:

```
$ npx hardhat test


  Voting Contract
```

```
    ✔ should have the correct owner after deployment
    ✔ should start an election with the correct parameters (63ms)
    ✔ should handle voter registration and voting (111ms)


  3 passing (2s)
```

Extended Testing Scenarios

An extended test file, `VotingExtended.test.js`, has been included in the test folder to cover 16 scenarios, providing a more comprehensive examination of the "Voting" smart contract. This set of tests delves into various situations and edge cases, offering a robust evaluation of the contract's behavior.

To run the extended tests, follow these steps:

1. Make sure you are located in the "voting-dapp-2023/hardhat" directory where your `hardhat.config.js` file is located.
2. Run the following command to execute the extended tests:

```
npx hardhat test test/VotingExtended.test.js
```

The extended test file includes scenarios such as:

- Verifying the correct owner after deployment.
- Testing the handling of voter registration and voting.
- Ensuring the proper ending/finalization of the election.
- Reinitializing the election after ending it.
- Verifying that additional votes are rejected if a voter has already voted.
- Testing the rejection of attempts to end the election by a non-administrator.
- Confirming that only the admin can register voters, start an election, reset an election, change the voting duration, and add candidates.
- Ensuring that only registered voters can cast votes.
- Restricting the admin from adding candidates after someone has voted.

These scenarios aim to cover a wide range of functionalities and potential scenarios that your "Voting" smart contract may encounter.

Feel free to review and run the extended tests to gain a deeper understanding of how your smart contract behaves in different situations.

Contact

S2BC