Blockchain & Solidity Lab2 – Crowdfunding 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 "CampaignCreator" Smart Contract

2.1. **Create a Test File:**

- Create a new file named `CampaignCreator.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");
require("@nomicfoundation/hardhat-toolbox");
const { ethers } = require("hardhat");
```

2.3. **Setup Test Environment:**

- Deploy the `CampaignCreator` contract before each test case.

```
describe("CampaignCreator", function () {
    let CampaignCreator;
    let campaignCreator;
    let deployer;

    beforeEach(async () => {
        [deployer] = await ethers.getSigners();

        CampaignCreator = await
ethers.getContractFactory("CampaignCreator");
        campaignCreator = await CampaignCreator.deploy();
    });
```

2.4. **Write Test Cases:**

- Write test cases to ensure the functionality of the `CampaignCreator` contract, such as deploying the contract and creating new campaigns.

```
it("should initially return an empty list of deployed campaigns", async
function () {
  const deployedCampaigns = await campaignCreator.getDeployedCampaigns();
  expect(deployedCampaigns).to.be.an("array").that.is.empty;
});
```

```
it("should create a new campaign with specified parameters and then check
if a campaign exists in the array of the contract", async function () {
  const minContribution = 1000;
  const description = "Test Campaign";

  await campaignCreator.createCampaign(minContribution, description);

  const deployedCampaigns = await campaignCreator.getDeployedCampaigns();
  expect(deployedCampaigns.length).to.equal(1);
});
```

2.5. **Running the Tests:**

- Execute the tests using the Hardhat CLI.

```
npx hardhat test
```

---

3. CampaignCreator.test.js

```
const { expect } = require("chai");
require("@nomicfoundation/hardhat-toolbox");
const { ethers } = require("hardhat");

describe("CampaignCreator", function () {
  let CampaignCreator;
  let campaignCreator;
  let deployer; // Declare deployer variable outside beforeEach to make it
accessible to other test cases

  beforeEach(async () => {
    console.log(
      "Before each hook executing... DEPLOY CampaignCreator smart
contract:",
      "\n"
    );
    [deployer] = await ethers.getSigners();

    CampaignCreator = await ethers.getContractFactory("CampaignCreator");
    campaignCreator = await CampaignCreator.deploy();

    // console.log(`json output: ${campaignCreator}`);
    // console.log("CampaignCreator object:");
    // console.dir(campaignCreator);

    // Extract runner address
    const Runner = campaignCreator.runner.address;
    console.log("Runner/Sender/Deployer Address:", Runner, "\n");
```

```
    // Extract target property
    const target = campaignCreator.target;
    console.log("Contract Target address:", target, "\n");
  });

  it("The contract got just deployed so it should return an empty list of
deployed campaigns initially", async function () {
    const deployedCampaigns = await campaignCreator.getDeployedCampaigns();
    expect(deployedCampaigns).to.be.an("array").that.is.empty;
  });

  it("should create a new campaign with specified parameters and then check
if a campaign exists in the array of the contract", async function () {
    const minContribution = 1000;
    const description = "Test Campaign";

    await campaignCreator.createCampaign(minContribution, description);

    const deployedCampaigns = await campaignCreator.getDeployedCampaigns();
    expect(deployedCampaigns.length).to.equal(1);

    // Add more assertions to verify the properties of the newly created
campaign

    // Get the address of the last deployed campaign
    const lastDeployedCampaignAddress =
      deployedCampaigns[deployedCampaigns.length - 1];

    // Log the address of the last deployed campaign
    console.log("Last deployed campaign address:",
lastDeployedCampaignAddress);
  });

  // Add more test cases as needed
});
```

4. Running the Tests

1. Make sure you are located in the 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 resemble the following:

```
$ npx hardhat test

  CampaignCreator
    ✔ should initially return an empty list of deployed campaigns (63ms)
    ✔ should create a new campaign with specified parameters and then
check if a campaign exists in the array of the contract (111ms)

  2 passing (2s)
```

Conclusion

Testing Ethereum smart contracts, such as the `CampaignCreator` contract, is essential for ensuring their reliability and security. By following best practices and writing comprehensive test cases, developers can identify and fix potential issues early in the development process, leading to more robust and secure blockchain applications.


```

This adapted documentation provides instructions for testing the `CampaignCreator` smart contract and includes code snippets for writing and running tests using Hardhat.

---

# Blockchain & Solidity Lab2 – Crowdfunding dApp Development

---

S2BC


```