Smart Contracts Exercise 01: Hello, Blockchain World!

Welcome to the first smart contracts exercise! In this first exercise, you will become familiar with the basics of smart contract development. The goal is to create a simple smart contract. You will then compile, test, and deploy this smart contract in the local network, and subsequently deploy it to the live blockchain.

1 Task: Set Up Foundry Environment

In this task, you will set up the Foundry development environment. Foundry is a fast, portable and modular toolkit for Ethereum application development. It consists of Forge (a testing framework), Cast (for interacting with EVM smart contracts), Anvil (for local Ethereum node) and Chisel (Solidity REPL). For this exercise, you can choose between using a Docker container or installing locally on your machine - select the option that best suits your development preferences.

1.1 Using Docker with VS Code

This option uses Docker to create a development environment with all the necessary tools and dependencies pre-installed.

Prerequisites:

- **Docker** - A platform for developing, shipping, and running applications in containers.
- **Visual Studio Code** - A lightweight but powerful source code editor.
- **Dev Containers** - An extension to VS Code that lets you use a Docker container as a full-featured development environment.

Setting Up the Project:

1. Visit the following GitLab repository and clone it to your local machine.
2. Open the repository folder in VS Code.
3. When prompted, click "Reopen in Container" or use the command palette (F1) and run `Dev Containers: Reopen in Container`.

Note: If you encounter permission issues when using Docker, you may need to adjust file permissions or run Docker with appropriate privileges. On Linux systems, you might need to add your user to the docker group: `sudo usermod -aG docker $USER` and then log out and back in.

1.2 Local Setup

If you prefer working directly on your machine without Docker, you can set up the development environment locally. Before setting up Foundry, ensure that you have the following installed on your system:

Prerequisites

- **Rust Toolchain** - Since Foundry is built in Rust, you'll need the Rust compiler and Cargo, Rust's package manager. The easiest way to install both is by using rustup.rs.
- **Bun** - JavaScript runtime & toolkit for installing dependencies and running scripts. Install it from bun.sh.

If you don't have Rust installed, you can install it using:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

To install Bun, use the following command:

```
$ curl -fsSL https://bun.sh/install | bash
```

Open your terminal and run the following command to verify the Rust and Bun installation:

```
$ rustc --version
$ cargo --version
$ bun --version
```

Installing Foundry

You can install Foundry using Foundryup, the official installer:

```
$ curl -L https://foundry.paradigm.xyz | bash
$ foundryup
```

This will install the Foundry toolkit, including:

- **Forge** - Testing framework for Ethereum
- **Cast** - Command-line tool for interacting with smart contracts
- **Anvil** - Local Ethereum node for development
- **Chisel** - Solidity REPL

Verify the installation by running:

```
$ forge --version
$ cast --version
$ anvil --version
```

- **Tip 1:** If you are using Windows, we strongly recommend using Windows Subsystem for Linux (WSL) to follow this guide. For more information, refer to the official documentation.
- **Tip 2:** If you are using Visual Studio Code, consider installing the Hardhat Solidity Extension. This extension helps your development process by providing features like syntax highlighting, code completion, etc.

1.3 Creating a New Foundry Project

Create an empty working directory and then run the following command to initialize a Foundry project:

```
$ forge init --no-git --force
```

`--force` Create the project even if the specified root directory is not empty.

`--no-git` Do not create a git repository.

This will create a new project in the current directory with the default template, which includes:

- `src/` - Directory for your smart contracts
- `test/` - Directory for your test files
- `script/` - Directory for your deployment scripts
- `foundry.toml` - Configuration file
- `lib/` - Directory for dependencies, with forge-std pre-installed

The initialization also creates example files: `Counter.sol` - Example smart contract, `Counter.s.sol` - Example deployment script, `Counter.t.sol` - Example test file. You can delete these example files if you want to start fresh.
For more information, see the forge init command documentation.

2 Task: Writing Your First Smart Contract

Start by creating a new file called `Greeter.sol` in the `src/` directory. Paste the code below into the file and take a minute to read the code.

```
// File: src/Greeter.sol

// SPDX-License-Identifier: MIT
pragma solidity 0.8.28; // Specify the Solidity compiler version

/**
 * @title Greeter
 * @dev A simple smart contract that stores a greeting message.
 */
contract Greeter {
    string private greeting; // State variable to store the greeting message

    /**
    * @dev Constructor that sets the initial greeting message upon deployment.
     * @param _greeting The greeting message to be stored.
     */
    constructor(string memory _greeting) {
        greeting = _greeting;
    }

    /**
     * @dev Function to retrieve the greeting message.
     * @return The current greeting stored in the contract.
     */
    function greet() public view returns (string memory) {
        return greeting;
    }
}
```

The Greeter contract is a simple Solidity smart contract that stores a greeting message, initializes it during deployment, and allows users to retrieve it via a public function. To compile the contract, run `forge build` in your terminal.

```
$ forge build
[] Compiling...
[] Compiling 2 files with Solc 0.8.28
[] Solc 0.8.28 finished in 252.53ms
Compiler run successful!
```

Forge compiled your Solidity smart contract and generated corresponding artifacts—including the contract's ABI (Application Binary Interface, which defines how to interact with the contract), bytecode (the compiled binary code that runs on the Ethereum Virtual Machine), and related metadata—which are stored in the `out` directory. You can take a look at the artifacts in `out/Greeter.sol/Greeter.json`.

3 Task: Test your Smart Contract with Forge

3.1 Writing a Simple Test

Create a new file called `Greeter.t.sol` in the `test/` directory with the following content:

```
// File: test/Greeter.t.sol

// SPDX-License-Identifier: MIT
pragma solidity 0.8.28;

// Import the contract to test
import {Greeter} from "../src/Greeter.sol";
// Import Forge's testing utilities
import {Test} from "forge-std/Test.sol";

// Test contract that inherits from Test
contract GreeterTest is Test {
    // Instance of the contract to test
    Greeter greeter;

    // The initial greeting for testing
    string constant INITIAL_GREETING = "Hello, Blockchain World!";

    // Set up function that runs before each test
    function setUp() public {
        greeter = new Greeter(INITIAL_GREETING);
    }

    // Test to ensure the initial greeting is set correctly
    function test_InitialGreeting() public view {
        string memory greeting = greeter.greet();
        assertEq(greeting, INITIAL_GREETING);
    }
}
```

In Foundry:

- Test contracts inherit from `Test` which provides utilities like assertions
- The `setUp()` function runs before each test (similar to "beforeEach" in other testing frameworks)
- Test functions start with `test_` prefix
- `assertEq()` and other assertion functions verify values match expected outcomes

3.2 Running the Test

Execute the test by running the following command in your terminal:

```
$ forge test
```

Expected output:

```
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Greeter.t.sol:GreeterTest
[PASS] test_InitialGreeting() (gas: 11982)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.01ms
(141.22µs CPU time)
```

```
Ran 1 test suite in 14.86ms (1.01ms CPU time): 1 tests passed,
0 failed, 0 skipped (1 total tests)
```

For more detailed output, including gas usage and execution traces, you can increase the verbosity by adding the `-v` flag:

```
$ forge test -vvvv
```

Congratulations! You wrote, compiled, and tested your first smart contract using Foundry!

4 Task: Deploying to a Live Network

Once you have programmed and tested your smart contract, you want to deploy it to a public blockchain so that others can access it. For the purposes of our exercise, we will not use the Ethereum mainnet because we would have to pay with real money, but instead use a live testnet. A testnet mimics real-world scenarios without risking our own money. Ethereum has several testnets; for our purposes, we will choose the Sepolia testnet. Deploying to a testnet is the same as deploying to mainnet at the software level. The only difference is the network you connect to.

4.1 Prerequisites

In order to finish this task, you will need the following tools:

- **MetaMask**: A popular Ethereum wallet that allows you to interact with the Ethereum blockchain. You can download the MetaMask extension for your browser from the official website and set it up. But you can also use other Ethereum wallets or simply create your own private-public key pair.
- **Infura API Key**: Infura provides access to Ethereum nodes without the need to run your own. Sign up at Infura to obtain an API key.
- **Sepolia Faucet**: Acquire Sepolia test Ether from a faucet to fund your deployment. Even on testnets, you'll need testnet ETH to pay for gas fees. Make sure you have enough Sepolia ETH (0.01 Sepolia ETH should be sufficent for this exercise) in your wallet before deployment. Gas prices fluctuate based on network congestion, even on testnets. Some reliable faucets include:
  - **Google Cloud Web3** (needs only Google account)
  - Metamask Sepolia Faucet (needs some ETH on mainnet)
  - Alchemy Sepolia Faucet (nedds some ETH on mainer)

4.2 Creating a Deployment Script

Foundry uses Solidity for deployment scripts. Create a new file called `DeployGreeter.s.sol` in the `script/` directory with the following content:

```
// File: script/DeployGreeter.s.sol

// SPDX-License-Identifier: MIT
pragma solidity 0.8.28;

// Import the contract to deploy
import {Greeter} from "../src/Greeter.sol";
// Import Forge's Script utilities
import {Script} from "forge-std/Script.sol";

// Script contract that inherits from Forge's Script
contract DeployGreeter is Script {
    function run() public returns (Greeter) {
        // Start recording transactions for deployment
        vm.startBroadcast();

        // Deploy the Greeter contract with initial message
        Greeter greeter = new Greeter("Hello, Blockchain World!");

        // Stop recording transactions
        vm.stopBroadcast();

        // Return the deployed contract
        return greeter;
    }
}
```

In Foundry:

- Deployment scripts inherit from the `Script` contract
- The `run()` function contains the deployment logic
- `vm.startBroadcast()` and `vm.stopBroadcast()` wrap the transactions that should be sent to the network

4.3 Configuring Foundry for Sepolia Deployment

To deploy your smart contract to the Sepolia testnet, you need to configure Foundry with the network details and your wallet credentials.

Storing Sensitive Information

It's crucial to keep sensitive information like your private key and Infura API key secure. We recommend using environment variables to manage these credentials only for the purpose of this exercise.

- Create a `.env` file in your project root:

```
SEPOLIA_RPC_URL=https://sepolia.infura.io/v3/YOUR_INFURA_API_KEY
PRIVATE_KEY=YOUR_PRIVATE_KEY_WITHOUT_0x_PREFIX
```

Replace the placeholders with your actual values. **Warning**: Never commit your `.env` file to version control. Add it to `.gitignore` to prevent accidental exposure of your private keys. Never use your private key associated with real money in plain text!

Updating `foundry.toml`

Modify your `foundry.toml` file to include the Sepolia network configuration:

```
[profile.default]
src = "src"
out = "out"
libs = ["lib"]
solc = "0.8.28"

[rpc_endpoints]
sepolia = "${SEPOLIA_RPC_URL}"
```

4.4 Deploying the Smart Contract to Sepolia

With the configuration in place, you're ready to deploy your smart contract to the Sepolia testnet.

```
$ source .env
$ forge script script/DeployGreeter.s.sol \
    --rpc-url sepolia \
    --private-key $PRIVATE_KEY \
    --broadcast
```

Expected Output:

```
[] Compiling...
[] Compiling 16 files with Solc 0.8.28
[] Solc 0.8.28 finished in 1.46s
Compiler run successful!
Script ran successfully.
== Return ==
0: contract Greeter 0xf4045726A4d561236b76AAeCD5ec80a02739B399
## Setting up 1 EVM.
==========================
Chain 11155111
Estimated gas price: 64.841239678 gwei
Estimated total gas used for script: 218697
Estimated amount required: 0.014180584593859566 ETH
==========================
##### sepolia
[Success] Hash: 0x8954a660307996cdca34039a07579ee35adf2cf8edc95bda5e0af74c4def0eac
Contract Address: 0xf4045726A4d561236b76AAeCD5ec80a02739B399
Block: 7901458
Paid: 0.005612595526892397 ETH (168229 gas * 33.362829993 gwei)
Sequence #1 on sepolia | Total Paid: 0.005612595526892397 ETH
(168229 gas * avg 33.362829993 gwei)
```

You can verify the deployment by visiting the Sepolia Etherscan explorer and searching for your contract address:
https://sepolia.etherscan.io/address/<ContractAddress>.
Search also for your account address and see your interactions with the deployed contract.

4.5 Interacting with Your Deployed Contract

Now that your contract is live on the Sepolia testnet, you can interact with it using Foundry's `cast` tool:

```
$ cast call <CONTRACT_ADDRESS> "greet()" --rpc-url sepolia
```

Expected Output:

```
0x0000000000000000000000000000000000000000000000000000000000000020000000000000000000 \
0000000000000000000000000000000000000000000000001848656c6c6f2c20426c6f636b636861696e \
20576f726c6421000000000000000000
```

To decode the response:

```
$ cast --to-ascii 0x0000000000000000000000000000000000000000000000000000000000000020000000000000000000 \
2000000000000000000000000000000000000000000000001848656c6c6f2c2042 \
6c6f636b636861696e20576f726c6421000000000000000000
```

Expected Output:

```
Hello, Blockchain World!
```

4.6 Further Reading

For more detailed information, refer to the following resources:

- Solidity Documentation
- Foundry Book
- Solidity by Example
- Forge Standard Library

Congratulations! You have successfully deployed your first smart contract to the live blockchain network! Stay tuned for the upcoming exercises!