Lab Session 4

# Theme of this Lab Session

Today Lab is split into two parts. This first part involves introducing you to basic and essential web development. This includes writing HTML file and using basic CSS to style or using available templates code available using the open-source CSS framework Bootstrap. The second part involves more advanced questions that are related to the lectures.

This particular Lab's setup is as follows: If you are highly experienced with HTML, CSS, and Bootstrap. Please sit in the back of the class with other colleagues and start working on part two of this Lab. Otherwise, please sit in front and team with another colleague and start working on part one of this Lab.

# Pre-requests skills for this Lab

* Having and running VS code editor with the configuration done in Lab session 1.

Main Lab Questions

# Part 1: Smart Contract Deployment using Remix

## Remix Integrated Development Environments (IDE) is an open-source web and desktop application that fosters fast development and has a rich set of plugins with intuitive GUIs. Furthermore, there is already a solidity compiler that is bundled to this IDE.[[1]](#footnote-2) This Part is tailored towards setting up Remix, learning how to use it by practice, creating your first solidity contract, deploying it, and interacting with it. You will learn how to interact with it using different blockchain: the web-version, local blockchain, and HTTP port (Metamask). Therefore, please answer the following set of questions.

## Questions:

1. Getting Started
   1. Go to [http://remix.ethereum.org](http://remix.ethereum.org/).
   2. Accept the popup that showed up.
   3. Observe the built-in plugins (icon panel showing icons) on the lefthand side. With one of your colleagues, comment on each plugin icon[[2]](#footnote-3).
2. Write your first smart contract[[3]](#footnote-4).
   1. Create A New File (Please follow the lecture videos)
      1. Click on the new file icon icon in the File Explorers icon panel on the left of the browser.
      2. Create a new file, name it "MyFirstContract.sol”. An empty file should have been formed
   2. Write the following code into the newly created file  
      
      1. What does the first line of the above code mean?
      2. What about the second line?
      3. What does the rest of the code mean?
3. Configure the compiler
   1. Press on the Solidity Compiler icon.
   2. Which version of the Solidity Compiler (Solc) should we select? Why?
   3. Press on the “Deploy & Run Transactions” icon
4. Deploying Smart Contract into the browser blockchain using JavaScript VM environment
   1. Select “JavaScript VM” on Remix
   2. Select an account
   3. In the “Contract”, make sure the correct Smart Contract (MyfirstContract.sol) is selected in the Dropdown
   4. hit "Deploy."
   5. Under “Deployed Contracts,” you should see that your first contract is deployed at an address.
   6. Press on this contract and you should see a button called “myreturn”.
   7. Hit this button
   8. What does it print to you?
5. Deploying Smart Contract into the Rinkeby testnet
   1. Connect MetaMask to Remix
      1. Sign in to your MetaMask (fox icon)
      2. Select “Injected Web3” on Remix
   2. Deploy the contract
      1. In the “Contract”, make sure the **correct** Smart Contract (MyfirstContract.sol) is selected in the Dropdown
      2. hit "Deploy" (this should trigger MetaMask to ask you if you really want to send this transaction.)
      3. Make sure the Rinkeby Test-Network is selected
      4. hit "Confirm"
   3. Check the contract account on EtherScan
      1. Go to the Rinkeby EtherScan [website](https://rinkeby.etherscan.io/).
      2. Navigate to the EOA address. Comment on what you see.
      3. What is the nonce of your Wallet? Hint: navigate to the transaction hash of the contract created.
6. Deploying Smart Contract into the local blockchain Ganache
   1. Open your terminal install ganache-cli “npm install ganache-cli” and write “ganache-cli”
   2. Go to Remix
   3. Select “Web3 Provider” on Remix
   4. Continue doing steps (c to h) of question 4.

# Part 2: Smart Contract Development using JavaScript

In the previous part, we did not use any JavaScript to interact with the EVM. However, as we know, JS is among the most popular languages in the Ethereum ecosystem. This is because it has many API libraries that make it easy for developers to interact with Ethereum. We have already been introduced to this language in Lab session three. However, there was no contract interaction involved. We only understood how to use JS libraries to understand further some concepts discussed in the lecture regarding externally owned accounts and how to use this language generally.   
  
This part is dedicated to interacting with Ethereum, mainly how a developer, with the help of JS API libraries, will be capable of deploying smart contracts to various EVM blockchains and interacting with these created smart contracts.

## Questions:

1. Setting up. Please follow the list to get started:
   1. Create a new folder call it “FirstContract”
   2. Open terminal at the folder
   3. Write “mkdir MyFirstContract”
   4. Move to the new folder “cd MyFirstContract”
   5. Initiate a package.json file “npm init -y”
   6. Install necessary nodejs packages “npm install solc web3 mocha ganache-cli @truffle/hdwallet-provider”
   7. Update your test script in the package.json file from “echo \"Error: no test specified\" && exit 1” to “test”:”mocha” (as done in the previous lab). Remember that mocha is an ES6 module.
   8. In the folder “MyFirstContract”, create the following.
      1. A “contracts” folder using “mkdir contracts”. We will need to store created solidity code smart contracts.
      2. A “test” folder using “mkdir test”. We will need this to store JavaScript test files that will test the compiled version of the solidity smart contract stored in the contracts folder.
   9. Open VS Code Editor using “code -n .”
      1. Go to “contracts”
      2. Create a new file
      3. Name it “MyFirstContract.sol”
      4. Copy the code from Remix and past it into his code, then save the file
   10. Back to the “Explorer” side and outside the contracts folder, create a comiler.js file.

Note: Now that you have set up a suitable environment for a demonstration on how to deploy a smart contract using JavaScript. The following questions will help you understand the details of such as setup and will act as a stepping stone to help you compile your solidity code.

1. In the compile.js, you will be using the following nodejs libraries [path](https://www.w3schools.com/nodejs/ref_path.asp), [fs](https://www.w3schools.com/nodejs/nodejs_filesystem.asp), and [solc](https://www.npmjs.com/package/solc?activeTab=readme). Therefore, please feel free to scan them. Accordingly, please answer the following questions.  
   1. Write a constant JS variable “Path1” that represents the path directory of your smart contract.
   2. Use the inbuilt application programming interface “readFileSync()” method of fs module to read the file whose path is “Path1” and return its content into a new constant variable called “source”. Remember that you should be using Unicode Transformation Format utf8[[4]](#footnote-5).
   3. Create an object variable called ‘jsonObj’ containing three keys: language, sources, settings.
      1. The key “language” should have the value of the language you are using, i.e. “Solidity”.
      2. The key “sources” represents an object that has an object key “MyFirstContract.sol” and this object has another key “content“ with a value source that of b.
      3. The key settings is an object that contains two objects and is mainly used for later usage in the solc package. Mainly this setting allows us to select all outputs the compiler can possibly generate. The code for this key object is available here  
           
         
   4. Using JSON.stringify() serialization the jsonObj object created in iii. Then, use the compile method of solc package and print the output.
   5. Using iv, JSON.parse(), and getting key values of an object. Try accessing the value of MyfirstContract.
   6. Export the output of v by simply assigning it to module.exports
2. In the previous lab session, we were unable to learn the necessary packages in JavaScript that helps you deploy and interact with your smart contract. The reason for this was that we were unable to reach the level we had just reached. That is, we learn how to write a simple solidity code and compile it with Nodejs. Now that we have this knowledge, it is an excellent time to learn how to deploy and interact with a smart contract. Notably, we shall be using the smart contract of this part along with everything we have done so far.  
   1. Deploying using Ganache.
      1. Do the following setup steps
         1. Create a JS file called “DeployTest.js” in the “MyFirstContract” folder
         2. As done in Lab session 3, import essential packages such as assert, ganache-cli, web3.
         3. Import the abi and evm keys from the outcome of the compiler.js file created in question 2 of part 2.
         4. Set an instance web3 such that Ganache is the provider
      2. Create an asynchronous function called “Deploying\_using\_sendTransaction”. In this function you must do the following
         1. Get accounts from ganache
         2. Create a transaction “tx” variable that contains all the data for deploying a smart contract mainly the “bytecode”
         3. Using the “sendTransaction” method of web3js send this transaction to the local blockchain using one of the accounts and store this variable as “results”
         4. Print the name of the contract address
         5. Try running this function
      3. The method "[Contract](https://web3js.readthedocs.io/en/v1.2.11/web3-eth-contract.html)" of web3.eth makes it easy to interact with smart contracts on the ethereum blockchain. Using this method, adjust the "Deploying\_using\_sendTransaction" function to call the deployed smart contract methods. Namely, getting back the "Hello World" string.
      4. Write another function called "TestDeploying\_using\_contract" that does exactly the same as "Deploying\_using\_sendTransaction". Use "Contract" and "deploy" methods of web3.eth instead of "sendTransaction" method.
   2. Deploying using Rinkeby testnet.
      1. In a new JS file, "DeployTestNet.js", and using the information you gained in part a of this question and the previous lab, rewrite the function "Deploying\_using\_sendTransaction" " to deploy your smart contract into the Rinkeby testnet.
      2. Record the contract public address.
      3. Using the abi from the compiler.js and the public address, write a function that allows you to call the myreturn() function written in solidity file. Call this function “InteractDeployedContract”
      4. Suppose you only have the contract address, can you interact with the contract without the abi?
      5. Write a function called "InteractDeployedContract1" whereby you will need to use the contract address only i.e. "0x71b6Bf4c8feb786A480df0E4C6FebAe1eBEc8587" to get the public message "Hello World".
      6. Why do you think the function " TestDeploying\_using\_contract” used in the Ganache case (part a) would be more complicated code than “Deploying\_using\_sendTransaction” for deploying your smart contract into the Rinkeby testnet?
3. In the previous question, we learn how to deploy and interact with a contract. However, this is the wrong strategy to go within an actual project. In fact, we usually test the code before deploying it. In this question, we will learn how to test out our smart contract before deploying.  
     
   The test folder is the folder that will store the testing javascript files. In the following set of questions, you should create a test for your solidity contract.  
   1. Setting-up
      1. Go to the test folder and create a JS file called “MyFirstContract.test.js”[[5]](#footnote-6) and make sure that all the packages required in question 3 are being imported to this file.
   2. Using the describe call, write an “it” call that tests
      1. whether you are capable of reading ganache accounts (hint: [Reference link](https://piyopiyo.medium.com/how-to-check-if-ethereum-address-is-valid-or-not-ef587b6c4819))
      2. whether the accounts read from ganache are addresses
      3. whether it can deploy a contract created in question 2.
   3. Run the test on the terminal when your current directory is MyFirstContract folder. Use the “npm run test”.[[6]](#footnote-7)
   4. You can see that each of the “it” call we needed to write “accounts = await web3.eth.getAccounts().” Which is cumbersome. However, Mocha has a [hook](https://mochajs.org/#:~:text=With%20its%20default%20%E2%80%9CBDD%E2%80%9D%2Dstyle%20interface%2C%20Mocha%20provides%20the%20hooks%20before()%2C%20after()%2C%20beforeEach()%2C%20and) known by “beforeEach”. Using this function before describe() rewrite your JS file.
   5. Using the “it” call test whether the public string from the deployed contract is exactly “Hello World”.

# Part 3: Smart Contract Development with Truffle

In the previous part, we learn how to deploy and interact with a basic smart contract using JavaScript. However, there is a better way of interacting and developing smart contracts. Today, [Truffle](https://trufflesuite.com/truffle) is widely considered the most popular tool for blockchain application development and therefore learning it is a must for being a smart contract developer. In this question, we shall learn how to handle smart contracts using Truffle. Particularly, many concepts from the previous exercise will prove helpful in Truffle, and in many instances, truffle documents will ask you to familiarize yourself with the concepts we were already discussing in the previous labs and the previous questions of this lab. It is now that we are ready to learn smart contract development using Truffle.

This part, therefore, aims at repeating the concepts learned in the previous part but using the truffle environment.

## Questions:

1. Steps required to set up truffle
   1. Install truffle globally on your machine by writing “npm install -g truffle” on your terminal
   2. Create a new folder “MyTruf” by writing on your terminal “mkdir MyTruf”
   3. Move to this working directory “cd MyTruf”
   4. To initialize a truffle project inside of your working directory write “truffle init” on your terminal. What did this command create? How does this compare to the set-up in the previous part?
   5. Compare the files available in this “MyTruf” to the folder created in the previous part (i.e. “MyFirstContract”). What does each of the files and folders created represent?
2. This question is dedicated to uploading your created smart contract and compiling it using truffle.
   1. In the contracts folder, delete the "Migrations.sol" and create the same solidity contract you had in the previous part of this lab “MyFirstContract.sol”.
      1. You can simply copy and past your previous solidity code or
      2. You can open your VS code editor using “code -n .” (windows) on the terminal. For mac users see this [link](https://stackoverflow.com/questions/30065227/run-open-vscode-from-mac-terminal).
   2. To compile one or all the solidity files available in the contracts folder, write "truffle compile" in the terminal. What changes have you seen in the folder “MyTruff”?
3. Deploy your smart contract into the local blockchain and do simple interactions using Truffle environment.
   1. Change the code in the file “1\_initial\_migration.js” located in the “migrations” folder so that the code deploys “MyFirstContract.sol” as opposed to “Migrations.sol”
   2. Before deploying your smart contract, it is crucial to set up the Ethereum client. Therefore, to do so, we need to fire up the local blockchain. Accordingly, in the terminal run, the “truffle develop”
   3. To deploy your contract, you simply need to run the code available in migrations folder. To do so, simply run “truffle migrate”
4. Interact with the deployed smart contract in Ganache using Truffle. In the previous question, you have set up the files and deployed your first contract into Ganache using truffle. In this question, you need to build upon the previous question to interact with your deployed smart contract using truffle. Therefore, it is expected that your terminal still has the “truffle(develop)”   
   1. Create an instance of your contract by writing the following “let instance\_contract = await MyfirstContract.deployed()”
   2. To call the function “myreturn()” from your smart contract (remember we wrote this in the MyFirstContract.sol) we could simply write “instance\_contract.myreturn()”. What do you expect the outcome to be?
5. Testing Smart Contract in Truffle environment.
   1. Stop the truffle development on your terminal by pressing ctrl+c several times
   2. Create a JS test file for your first contract “MyFirstContract.sol” this can be done by
      1. On the terminal
         1. Write “npm install -g touch-for-windows” to install touch package
         2. Write “touch test\MyFirstContract.test.js”
      2. On VS code editor
   3. Open this file and do the following on VS code “code -n .”
      1. Import your smart contract using artifact.require()
      2. Instead of using describe() function use contract()
      3. Write a code that tests whether the outcome of the deployed function is “Hello Word” as we expect it to be.
   4. Run the test on the terminal “truffle test”

# Part 4: Tutorial Questions

1. Environment refers to the collection of hardware and software tools a system developer uses to build software systems using the remix [documentation](https://remix-ide.readthedocs.io/en/latest/run.html#:~:text=to%20Deploy%20%26%20Run.-,Environment,-JavaScript%20VM%3A%20All). How many different environments are there, and what are the differences between each environment?
2. When you deployed your first contract in Remix, under “Deployed contracts”, you would have seen an address. What does this address represent, and how was it computed?
3. In Part 2 of this lab, you were asked to create a “compile.js” file. What is its role?
4. In the truffle initialization, a migrations folder was generated. In this folder, there is a file “1\_initial\_migration.js”. Using the Truffle webpage and your own narrative, what does each line mean? Please illustrate your answer with examples used within this Lab.

# Inspired References

* Mastering Ethereum Building Smart Contracts and Dapps
* GeeksforGeeks website
* Shared links

1. For further information about Remix please consult the [Remix document](https://readthedocs.org/projects/remix-ide/downloads/pdf/latest/). [↑](#footnote-ref-2)
2. If the plugins are not showing up yet, then click on the plugin symbol and enable them. If you do not find it, please feel free to raise your hand so that one of the teaching team member can support you and provide you with feedback. [↑](#footnote-ref-3)
3. Here we are following Chapter 4, with minor updates and changes, of "Hands-On Smart Contract Development with Solidity & Ethereum From Fundamentals to Deployment". Therefore, please feel free to read this book. The updates are implemented by using the latest solidity version. Currently, the latest solidity version is 0.8.10; therefore, please have a read of its [documentation](https://docs.soliditylang.org/en/v0.8.10/). [↑](#footnote-ref-4)
4. A good video that explains the concept of Unicode is available [here](https://www.youtube.com/watch?v=5aJKKgSEUnY). [↑](#footnote-ref-5)
5. Usually, it is a good practice to name your testing file a meaningful name. I called my testing file "MyFirstContract.test.js" because we are about to test our solidity contract known by MyFirstContract. [↑](#footnote-ref-6)
6. Npm run test calls the command specified in the scripts. test property of package.json, if you remember we put that to be mocha. [↑](#footnote-ref-7)