# DA51 Lab session 3

# Ethereum
# The Life Cycle of a Smart Contract.

J. Gaber, gaber@utbm.fr

**Target set of this session:**
After completion of this session, you will successfully know:

- how to write a smart contract using Solidity
- how to use the solc compiler to generate Ethereum bytecode
- how to script smart contract compilation and define the types of output required, including bytecode and ABI.
- how to write test cases by using Mocha, group them into a test suite, and describe a common set of actions to run before each test case.
- you will know how the steps for building, testing, and deploying a smart contract can be scripted for the development of your future distributed application.
- you will become familiar with using the Truffle Suite for smart contract development and lifecycle management.

**Developing a Contract with Solidity**

In the previous lab, you have created a private Ethereum network containing two nodes, and then you created accounts belonging to each of them. You also initiated transactions to transfer ether from one of those accounts to another.

In this lab, we will make use of Node.js and a solidity compiler in order to build out our smart contracts.

1. if you do not have Node.js already, you can navigate to nodejs.org and then from here, select the installation for your own operating system.

2. To test the installation, bring up a terminal session and execute the command node -v to print out the version of Node which has been installed.

3. You will need to make use of NPM (Node Package Manager), which comes packaged with Node.js. Check its version with the command npm -v

4. Recall the purpose of NPM?

You can at this step begin the development of your smart contract.

5. create a new project directory to create a new JavaScript project, and call it Lab3

6. use the command npm init to lay out a structure for the project. Running this command will prompt you to enter details for your project, you can keep hitting Enter to use the default values.

7. One the command is executed, examine the command output feedback and check your directory. What do you get?

package.json file is a project descriptor file (similar to pom.xm or build.gradle if you have worked with Maven or Gradle before). It contains project metadata and also will includes the dependencies of the project (we will add later a few dependencies in this lab).

8. You can look at its contents using cat or more utility. What is the list of information displayed as output?

In what follows, to begin development, you can use any IDE or text editor of your choice for this purpose, such as Sublime text editor or visual code (check this link https://code.visualstudio.com/docs/remote/wsl-tutorial if you are using wsl).

9. From your favorite IDE or otherwise you head back to the Lab3 directory and create a new source folder, which will store the smart contract, call it contracts, and then create a new source file within it.

10. The contract you will create contains a single field which is of type string and the name of the string variable is Message, which can be initialized using a constructor. The value can be set or retrieved by calling Setter/Getter functions.

```
1.  // SPDX-License-Identifier: UTBM

2.  pragma solidity >=0.8.0 <0.9.0;

3.  contract MyLab3SContract {
4.      string message;

5.      constructor (string memory initialMessage) {
6.          message = initialMessage;
7.      }

8.      function setMyLab3SContract (string memory newMessage) public {
9.          message = newMessage;
10.     }

11.     function getMyLab3SContract() public view returns (string memory) {
12.         return (message);
13.     }
14. }
```

11. Save the file as Bonjour.sol, where sol is the extension for a solidity source file.

12. walk through the code to get the interpretation of each line according to the solidity programming language.

**Compiling a Solidity Smart Contract**

You wrote out the code for your first smart contract in the solidity programming language. In what follows, you will compile this code and then view the output of the compiler.

To compile the smart contract, you will need to download and install the solc compiler.

13. Run the following command: sudo npm install solc -g

14. Why npm is used in the command sudo npm install solc -g?

you will be able to compile smart contracts by running solcjs.

However, if you would like to use the official solidity compiler using the solc binary rather than solcjs, then you will need to get this from the ethereum repo.

15. Run the command: brew install solidity. Note that you may need first to add Ethereum repo with the command: brew tap ethereum/ebrew install solcthereum (as you did in Lab2)

16. Run solc from your shell to view what the options are, in particular, the sections Output Components and Exta Output

17. Execute solc MyLab3SContract.sol

If there are no compile errors that you run into, you get successful feedback. And the output indicates there was no output which was generated. this is because you need to explicitly specify the kind of output you need from the compile step.

rerun solc, but this time you include the option combined-json and you specify the types of output you need.

18. Run the command: solc --combined-json=abi,bin,metadata --output-dir . MyLab3SContract.sol

19. What did you include in the output you expect?

Once you have executed this statement you see that the compiler run was once more successful, and some artifacts have been generated.

20. List (by running ls under WSL) to check the output

21. Which file is showed in addition to our solidity source file?
There is also a combined.json file

22. View the content of the file combined.json (by running cat or more under WSL)

23. Which data or fields are included in this file?
It includes the three different output formats, which we had specified. It includes the following fields: abi, bin, and metadata

24. What is the abi output?
There is the abi output which is a descriptor of the functions and the data within a contract which will allow us to interact with it.

25. What is the bin output?

Then there is the binary byte code for the contract.

26. What does the metadata field include?
And finally, there is a metadata field which includes various details about the contract itself.

This output is not easy to read. You can examine it with a JSON formatter.

27. Copy over the entire json output and then navigate your browser to the free online json formatter: jsonformatter.curiousconcept.comcat

28. Paste the copied output in the input field and clicks the Process button.

You can see that there is an interface which allows you to collapse and expand fields in your contract. A full screen view is also available.

29. You can examine the filed that contains the abi output.

30. You can examine the filed that contains the bin output.

31. And the filed that contains the metadata output.

32. You can also see the version of Solidity which was used to compile the source file.

**Writing a Compile Script**

So far, you wrote out a solidity smart contract and then compile it using the solc command line interface.

In the context of a more complex compile step, you define code for it. You can add some dependencies and you can automate testing and deployment as well.
The package.json file, which is the project descriptor, should include all of the dependencies for the project. There are no dependencies yet.

You will run npm install command to install some required packages and add as dependencies. This includes the solc solidity compiler, the mocha testing framework, the ganache-cli, which will help you to create a private network in which to deploy a smart contract, and the web3 package, which will allow you to interact with your Ethereum network and the smart contract.

33. Run the following command: npm install --save solc mocha ganache-cli web3

34. Check that these four dependencies have now been added to the project descriptor file?

In addition to this change, a new package-lock.json file has also been created and there is also a node_modules directory.

35. Examine the content of the file package-lock.json

36. Examine the content of node_modules subdirectory web3 for example

37. You will now define your compile procedure. In the root directory of the App, create a new file and call it compile.js.

38. Here is the entire contents of this compile script:

```
1.  const path = require('path');
2.  const fs = require('fs');
3.  const solc = require('solc');

4.  const MyLab3SContractPath = path.resolve(__dirname, "contracts", "MyLab3SContract.sol");

5.  const compilerInput = {
6.      language: "Solidity",
7.      sources: {
8.          'MyLab3SContract': { content: fs.readFileSync(MyLab3SContractPath, 'utf8') }
9.      },

10.       settings: {
11.       outputSelection: {
12.         "*": {
13.           "*": [ "abi", "evm.bytecode" ]
14.         }
15.       }
16.     }
17. };

18. console.log(solc.compile(JSON.stringify(compilerInput)));
19. // module.exports = solc.compile(JSON.stringify(compilerInput).conracts[':evm'])
20. //module.exports = solc.compile(JSON.stringify(compilerInput));
```

Line 1-3: the modules which you will require in this compile script includes the path, fs, and solc modules and each of these will be imported using the require function, referenced by using three different constants, path, fs, and solc.

Line 4: A constant to reference the path to our Solidity source file. The _dirname variable references the directory in which the compile.js file is present, and from there the source file at the subdirectory contracts/MyLab3SContract.sol

Line 5-9: define the input to your compiler as a JSON object, called compiler input, with various fields: the language of the source file, i.e., solidity, the source within the sources object, with its path to the file's system method readFileSync.

Line 10 -16: define what the output is going to be, i.e., you specify what the compiler will produce: an abi output, and the evm.bytecode of the smart contract.

Line 18: the output of the compile step by calling solc.compile, with the JSON version of your compiler input definition, will be printed out to the console.

39. Run the command: node compile.js from the terminal to invoke your compile script.

40. Since the output generated is rather large and difficult to view, copy this JSON output and then head over to the JSON formatter page once more to view it much easier.

41. Check the binary bytecode available within evm.object, evm.opcodes gives you the full opcodes of the smart contract.

**The Mocha Test Framework integration with Solidity and Ethereum**

So far, you have seen how you can automate various steps in a build by making use of a compile script.

In what follows, you will see how we can make use of the Mocha testing framework to perform automated testing of a Solidity smart contract.

42. Make a modification to your compiler script, instead of printing out the compiler output to the console, you will export the compiler output so that it can be used by your testing script. Put comment mark on line 18 and remove comment from Line 20.

43. Run the command: node compile.js

44. Did you get an output on the console? why?

Adding the compiler output to module.exports will ensure that another JavaScript file which you will create will be able to reference the compiler output by making use of the require function.

45. create a new folder within the App directory, and name it test.

46. In this folder, create a new file and call it MyLab3SContract.test.js

47. Here is the code to include in this file:

```
1.  const assert = require('assert');
2.  const ganache = require('ganache-cli');
3.  const Web3 = require('web3');
4.  const web3 = new Web3(ganache.provider());
5.  //const web3 = new Web3('http://localhost:8545');
6.  //const web3 = new Web3(ganache.provider(), null, { transactionConfirmationBlocks: 1 });
7.  //const web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));

8.  let accounts;
9.  beforeEach(async () => {
10.         accounts = await web3.eth.getAccounts()

11. });

12. describe('MyLab3SContract', () => {
13.         it('Deploy a contract', () => {
14.                 console.log(accounts)
15.         });
16. });
```

Line 1-4: importing all the required packages with require function calls: Assert, Ganache CLI, and the web3 package. you will integrate the test with Ethereum by making use of the web3 package. The Ethereum network you will be working with is a private network which has been provisioned using the Ganache framework. You initialize the web3 interface with a

private network which would be spun up by Ganache. Recall that when Ganache spins up its private network, it creates 10 different accounts and each of them are loaded with 100 ether.

Line 8-11: within beforeEach method, you define an operation which needs to be performed before each of the individual tests to perform. And in this case, you fetch all of the accounts which are available via Ganache.

Line 12-16: to group various tests together. This group of tests will be called 'MyLab3SContract'. It could include several different test cases. Each individual test case definition can be found within the it functions. There is one test case here. It is called ‘Deploy a contract’ and print out all of the account addresses which have been retrieved on the console.

48. You need to ensure that Mocha will be that framework which is used to execute the test script. Head over to the package.json file, and within the scripts field, you must configure your test script to use the Mocha testing framework in order to execute its tests, by changing:

```
"scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
},
```

to

```
"scripts": {
   "test": "mocha"
  },
```

49. Make sure also that within the dependencies section, the Mocha package has been included.

50. To execute the test script, you can head over to the shell, and from the root project directory, run the command: npm test.

You get the result that the test case which you have defined has passed:

```
gaber@PCP-GI-07:~/Blockchain/test$ npm test

> test@1.0.0 test
> mocha

  MyLab3SContract
[
  '0xB6E0a844ff8729dAD545c59A4B88F00D9C43eBb7',
  '0x7093D8cc5EC973fa4637206D683748A2ac8014ac',
  '0x3a91155ac608F0B5214c5AB1409738c13d86Fd8f',
  '0x019eF3f0C76aEE81D33A9eE2840fb07965705227',
  '0xae4726c5037BA20ED827Db45f1D934788634E181',
  '0x3AeBB13197cAb2180eFf4815183dE4c9DA689074',
  '0xcce45BE2989f9e593D4980180CD5aace82F5eceD',
  '0x4F573FfcF7D00Ba64cC25D43e02375D8Fd5b6551',
  '0x8F9001d573503d53caFc60bC01D616ecb972616d',
  '0x586BBF31066f0A98457447657cb4B60b31b311D8'
]
    ✔ Deploy a contract


  1 passing (69ms)
```

gaber@PCP-GI-07:~/Blockchain/test$

It is the addresses of those 10 accounts which you will see in your output, and you can also confirm that the deploy a contract test case has passed as the green tick is displayed.