15

Introducing Web3

Web3 is a JavaScript library that can be used to communicate with an Ethereum node via RPC communication. Web3 works by exposing methods that have been enabled over RPC. This allows the development of **user interfaces** (**UIs**) that make use of the Web3 library in order to interact with contracts deployed over the blockchain.

In this chapter, we'll explore the Web3 API, and introduce some detailed examples of how smart contracts are written, tested, and deployed to the Ethereum blockchain. We will use various tools such as the Remix IDE and Ganache to develop and test smart contracts, and look at the methods used to deploy smart contracts to Ethereum test networks and private networks. The chapter will explore how HTML and JavaScript frontends can be developed to interact with smart contracts deployed on the blockchain, and introduce advanced libraries such as Drizzle. The topics we will cover are as follows:

- Exploring Web3 with Geth
- Contract deployment
- Interacting with contracts via frontends
- Development frameworks

We will start with Web3, and gradually build our knowledge with various tools and techniques for smart contract development. You will be able to test your knowledge in the bonus content pages for this book, where we will develop a project using all the techniques that this chapter will cover.

Exploring Web3 with Geth

In order to expose the required APIs via `geth`, the following command can be used:

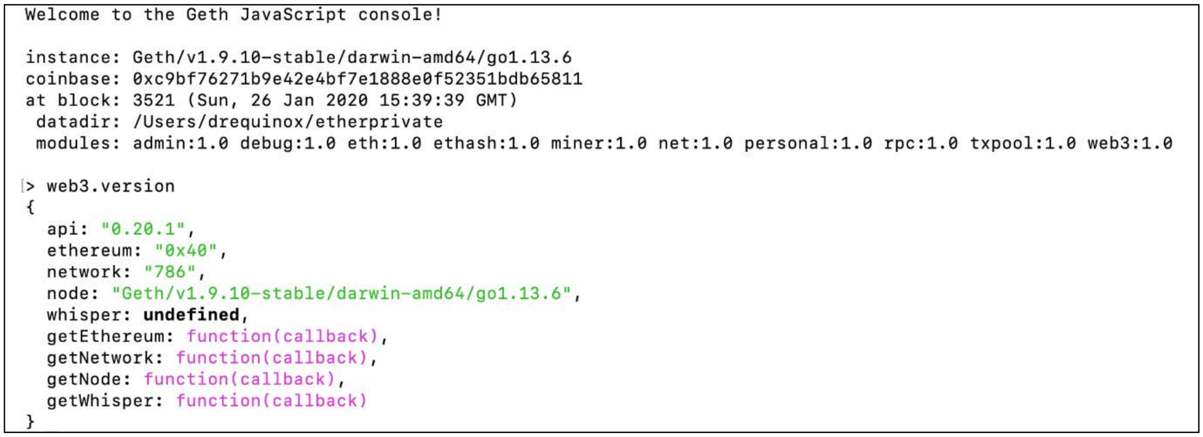
```
$ geth --datadir ~/etherprivate --networkid 786 --rpc --rpcapi
"web3,net,eth,debug" --rpcport 8001 --rpccorsdomain http://localhost:7777
```

> The `--rpcapi` flag in the preceding command allows the `web3`, `eth`, `net`, and `debug` methods. There are other APIs such as `personal`, `miner`, and `admin` that can be exposed by adding their names to this list.

Web3 is a powerful API and can be explored by attaching a `geth` instance. Later in this section, you will be introduced to the concepts and techniques of making use of Web3 via JavaScript/HTML frontends. The `geth` instance can be attached using the following command:

```
$ geth attach ~/etherprivate/geth.ipc
```

Once the `geth` JavaScript console is running, Web3 can be queried:



```
Welcome to the Geth JavaScript console!

instance: Geth/v1.9.10-stable/darwin-amd64/go1.13.6
coinbase: 0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811
at block: 3521 (Sun, 26 Jan 2020 15:39:39 GMT)
 datadir: /Users/drequinox/etherprivate
 modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

> web3.version
{
  api: "0.20.1",
  ethereum: "0x40",
  network: "786",
  node: "Geth/v1.9.10-stable/darwin-amd64/go1.13.6",
  whisper: undefined,
  getEthereum: function(callback),
  getNetwork: function(callback),
  getNode: function(callback),
  getWhisper: function(callback)
}
```

Figure 15.1: Web3 via geth.ipc

Now that we've introduced Web3, let's consider how the Remix IDE can be used to deploy a contract, and how the `geth` console can interact with the deployed contract.

Contract deployment

A simple contract can be deployed using Geth and interacted with using Web3 via the **command-line interface** (**CLI**) that `geth` provides (`console` or `attach`). The following are the steps to achieve that. As an example, the following source code will be used:

```
pragma solidity ^0.4.0;
contract valueChecker
{
    uint price=10;
    event valueEvent(bool returnValue);
    function Matcher (uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

Run the geth client using the following command:

```
$ geth --datadir ~/etherprivate --networkid 786 --allow-insecure-unlock -rpc
--rpcapi "web3,net,eth,debug,personal" --rpccorsdomain https://remix.ethereum.org
--debug --vmdebug -nodiscover
```

or

```
$ geth --datadir ~/etherprivate --networkid 786 --allow-insecure-unlock
--rpc --rpcapi "web3,net,eth,debug" --rpcport 8001 --rpccorsdomain "http://
localhost:7777"
```

Alternatively, the following command can be run:

```
$ geth --datadir ~/etherprivate/ --networkid 786 --rpc --rpcapi
'web3,eth,net,debug,personal'  --rpccorsdomain '*'
```

We've used all these commands in different contexts before, and you can use any of them to run Geth. The difference is that the first command allows connectivity with the Remix IDE as we have specified `--rpccorsdomain https://remix.ethereum.org`. If you need to use the Remix IDE then use the first command.

The second command allows `localhost:7777` to access the RPC server exposed by `geth`. This option is useful if you have an application running on this interface and you want to give it access to RPC. It also exposes RPC on port `8001` via the flag `--rpcport 8001`, which is useful in case you have some other service or application listening already on port `8545`, which would means that `geth` won't be able to use that already-in-use port. This is because Geth listens on port `8545` for the HTTP-RPC server by default.

Alternatively, you can simply run the last command, which allows all incoming connections as `--rpccorsdomain` is set to `*`.

If the Geth console is not already running, open another terminal and run the following command:

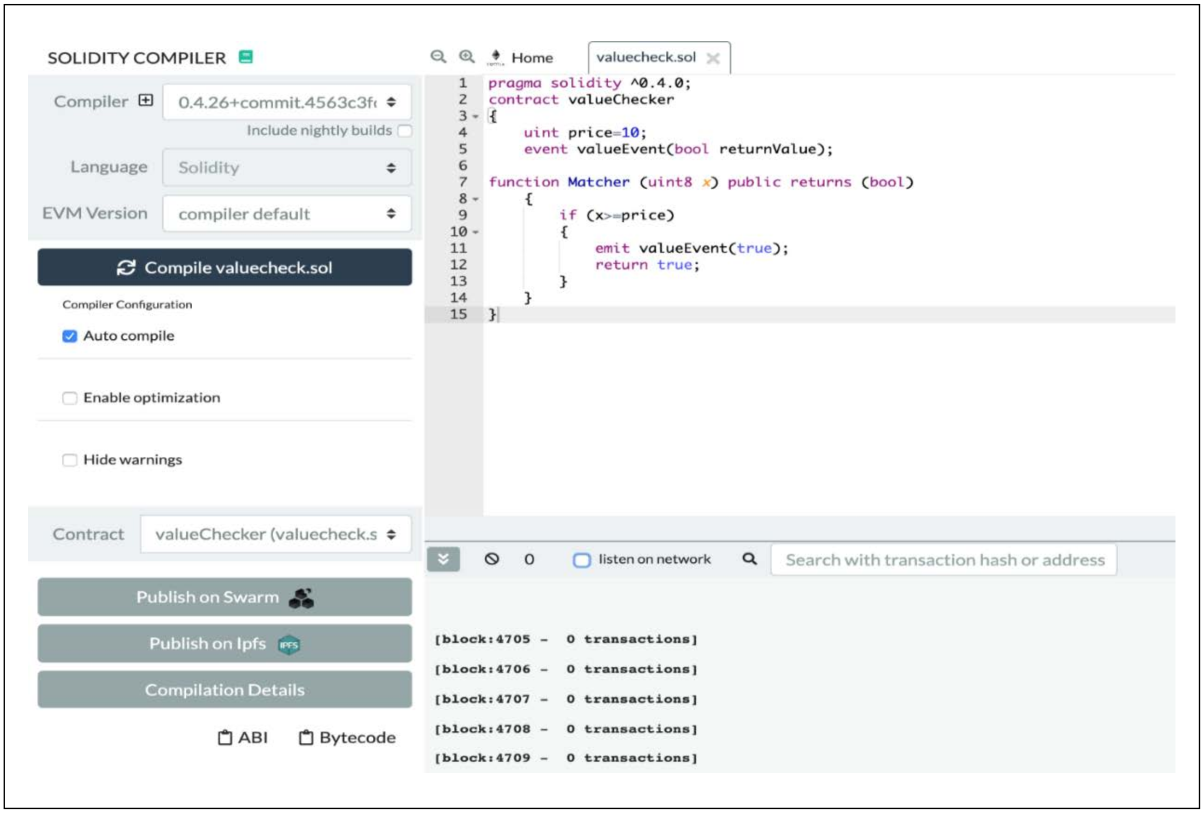
```
$ geth attach ~/etherprivate/geth.ipc
```

In order to deploy a smart contract, we use the Web3 deployment script. The main elements of the script, the **Application Binary Interface** (**ABI**) and the bytecode, can be generated from the Remix IDE. To learn how to download and use the Remix IDE, refer to *Chapter 13*, *Ethereum Development Environment*. First, paste the following source code, as mentioned at the beginning of the section, in the Remix IDE.

We will discuss the Remix IDE in more detail later in this chapter; for now, we are using this IDE only to get the required elements (ABI and bytecode) for the Web3 deployment script used for deployment of the contract:

```
pragma solidity ^0.4.0;
contract valueChecker
{
    uint price=10;
    event valueEvent(bool returnValue);
    function Matcher (uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

Once the code is pasted in the Remix IDE, it will appear as follows in the Solidity compiler:



Compiler: 0.4.26+commit.4563c3f
Include nightly builds
Language: Solidity
EVM Version: compiler default
Compiler Configuration
Auto compile ✓
Enable optimization
Hide warnings
Contract: valueChecker (valuecheck.s

```
pragma solidity ^0.4.0;
contract valueChecker
{
    uint price=10;
    event valueEvent(bool returnValue);

function Matcher (uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

```
[block:4705 - 0 transactions]
[block:4706 - 0 transactions]
[block:4707 - 0 transactions]
[block:4708 - 0 transactions]
[block:4709 - 0 transactions]
```

Figure 15.2: Code shown in Remix

Now create the Web3 deployment script as shown as follows. First generate the ABI and bytecode by using the **ABI** and **Bytecode** buttons in the Remix IDE (shown at the lower-left side in *Figure 15.2*) and paste them in the following script after the `web3.eth.contract(` and `data:` elements, respectively:

```
var valuecheckerContract = web3.eth.contract([{"anonymous":false,"inputs":[{"inde
xed":false,"internalType":"bool","name":"returnValue","type":"bool"}],"name":"val
ueEvent","type":"event"},{"inputs":[{"internalType":"uint8","name":"x","type":"ui
nt8"}],"name":"Matcher","outputs":[{"internalType":"bool","name":"","type":"bool"
}],"stateMutability":"nonpayable","type":"function"}]);
var valuechecker = valuecheckerContract.new(
   {
     from: web3.eth.accounts[0],
     data: '0x6080604052600a600055348015610015576000080fd5b5061010d806100256000396000
00f300608060405260043610603f576000357c010000000000000000000000000000000000000000000
00000000000000000900463ffffffff168063f9d55e21146044575b600080fd5b348015604f57600080fd5b50
606f600480360381019080803560ff16906020019092919050505060895 65b6040518082151515158 1
52602001915050604051809103 90f35b600080548260ff1610151560db577f3eb1a229ff7995457774a
4bd31ef7b13b6f4491ad1ebb8961af120b8b4b6239c60016040518082151515158152602001915050 6
0405180910390a16001905060dc565b5b9190505600a165627a7a723058209ff756514f1ef46f5650d8
00506c4eb6be2d8d71c0e2c8b0ca50660fde82c7680029',
     gas: '4700000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + '
transactionHash: ' + contract.transactionHash);
    }
})
```

Ensure that the accounts are unlocked. First list the accounts by using the following command, which outputs account `0` (first account), as shown here:

```
> personal.listAccounts[0]
"0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811"
```

Now unlock the account using the following command. It will need the passphrase (password) that you used originally when creating this account. Enter the password to unlock the account:

```
> personal.unlockAccount(personal.listAccounts[0])
Unlock account 0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811
Password:
true
```

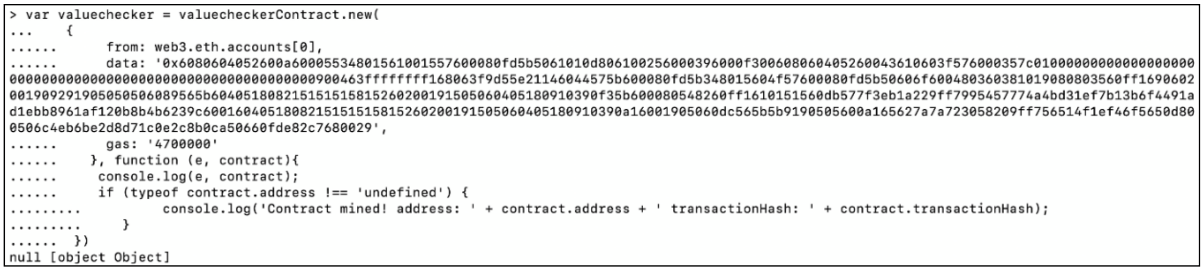
For more convenience, the account can be unlocked permanently for the length of the `geth` console/attach session by using the command shown here.

```
> web3.personal.unlockAccount("0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811",
"Password123", 0);
true
```

Now, we can open the `geth` console that has been opened previously, and deploy the contract. However, before deploying the contract, make sure that mining is running on the `geth` node. The following command can be used to start mining under the `geth` console.

```
> miner.start()
```

Now paste the previously mentioned Web3 deployment script in the `geth` console as shown in the following screenshot:



```
> var valuechecker = valuecheckerContract.new(
...     {
......      from: web3.eth.accounts[0],
......      data: '0x6080604052600a600055348015610015576000080fd5b5061010d806100256000396000f300608060405260043610603f576000357c0100000000000000000000
000000000000000000000000000000000000000900463ffffffff168063f9d55e21146044575b600080fd5b348015604f57600080fd5b50606f60048036038101908080803560ff1690602
0019092919050505060089565b604051808215151515815260200191505060405180910390f35b600080548260ff1610151560db577f3eb1a229ff7995457774a4bd31ef7b13b6f4491a
d1ebb8961af120b8b4b6239c600160405180821515151581526020019150506040518091039 0a16001905060dc565b5b9190505600a165627a7a723058209ff756514f1ef46f5650d80
0506c4eb6be2d8d71c0e2c8b0ca50660fde82c7680029',
......      gas: '4700000'
......    }, function (e, contract){
......     console.log(e, contract);
......     if (typeof contract.address !== 'undefined') {
.........         console.log('Contract mined! address: ' + contract.address + ' transactionHash: ' + contract.transactionHash);
.........     }
......  })
null [object Object]
```

Figure 15.3: The Web3 deployment script deployment using Geth

The previous screenshot shows the output as it looks when the Web3 deployment script is pasted in the Geth console for deployment. Note the data (code) section, which contains the bytecode of the smart contract.

ABI and code can also be obtained by using the Solidity compiler, as shown in the following code snippets.

In order to generate the ABI we can use the command shown as follows:

```
$ solc --abi valuechecker.sol
```

This command will produce the following output, with the contract ABI in `JSON` format:

```
======= valuechecker.sol:valueChecker =======
Contract JSON ABI
[{"anonymous":false,"inputs":[{"indexed":false,"internalType":"bool",
"name":"returnValue","type":"bool"}],"name":"valueEvent","type":"event"}
,{"inputs":[{"internalType":"uint8","name":"x","type":"uint8"}],"name":
"Matcher","outputs":[{"internalType":"bool","name":"","type":"bool"}]
,"stateMutability":"nonpayable","type":"function"}]
```

The next step is to generate code, which we can use the following command to do:

```
$ solc --bin valuechecker.sol
```

This command will produce the binary (represented as hex) of the smart contract code:

```
======= valuechecker.sol:valueChecker =======
Binary:
6080604052600a600055348015610015576000080fd5b5061010d806100256000396000
0f3006080604052600436106003f576000357c0100000000000000000000000000000000
000000000000000000000000000900463ffffffff168063f9d55e21146044575b60008
0fd5b348015604f57600080fd5b50606f6004803603810190808035600ff1690602001
909291905050506089565b6040518082151515158152602001915050604051809103 9
0f35b600080548260ff161015156 0db577f3eb1a229ff7995457774a4bd31ef7b13b6
f4491ad1ebb8961af120b8b4b6239c60016040518082151515158152602001915050 6
0405180910390a16001905060dc565b5b9190505600a165627a7a723058209ff75651
4f1ef46f5650d800506c4eb6be2d8d71c0e2c8b0ca50660fde82c7680029
```

We can also see the relevant message in the `geth` logs to verify that the contract creation transaction has been submitted; you will see messages similar to the one shown as follows:

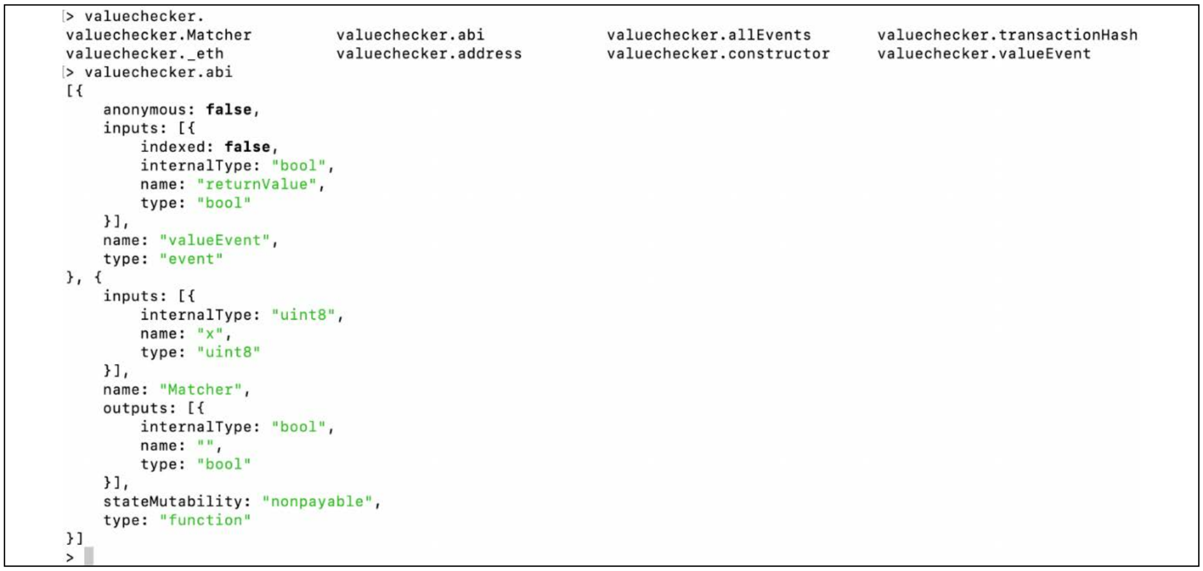
```
INFO  . . . . . Submitted contract creation
fullhash=0x73fcceace856553513fa25d0ee9e4a085e23a508e17ae811960b0e28a1
98efab contract=0x82012b7601Fd23669B50Bb7Dd79460970cE386e3
```

Also notice that in the Geth console, the following message appears as soon as the transaction is mined, indicating that the contract has been successfully mined:

```
Contract mined! address: 0x82012b7601fd23669b50bb7dd79460970ce386e3
transactionHash:
0x73fcceace856553513fa25d0ee9e4a085e23a508e17ae811960b0e28a198efab
```

Notice that in the preceding output, the transaction hash `0x73fcceace856553513fa25d0ee9e4a085e23a508e17ae811960b0e28a198efab` is also shown.

After the contract is deployed successfully, you can query various attributes related to this contract, which we will also use later in this example, such as the contract address and ABI definition, as shown in the following screenshot. Remember, all of these commands are issued via the `geth` console, which we have already opened and used for contract deployment.



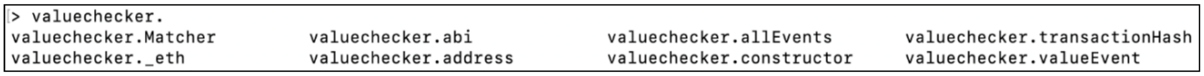
```
> valuechecker.
valuechecker.Matcher        valuechecker.abi            valuechecker.allEvents      valuechecker.transactionHash
valuechecker._eth           valuechecker.address        valuechecker.constructor    valuechecker.valueEvent
> valuechecker.abi
[{
    anonymous: false,
    inputs: [{
        indexed: false,
        internalType: "bool",
        name: "returnValue",
        type: "bool"
    }],
    name: "valueEvent",
    type: "event"
}, {
    inputs: [{
        internalType: "uint8",
        name: "x",
        type: "uint8"
    }],
    name: "Matcher",
    outputs: [{
        internalType: "bool",
        name: "",
        type: "bool"
    }],
    stateMutability: "nonpayable",
    type: "function"
}]
>
```

Figure 15.4: Value checker attributes

In order to make interaction with the contract easier, the address of the account can be assigned to a variable. There are a number of methods that are now exposed, and the contract can be further queried now, for example:

```
> eth.getBalance(valuechecker.address)
0
```

We can now call the actual methods in the contract. A list of the various methods that have been exposed now can be seen as follows:



```
> valuechecker.
valuechecker.Matcher        valuechecker.abi            valuechecker.allEvents      valuechecker.transactionHash
valuechecker._eth           valuechecker.address        valuechecker.constructor    valuechecker.valueEvent
```

Figure 15.5: Various methods for valuechecker

The contract can be further queried as follows.

First, we find the transaction hash, which identifies the transaction:

```
> valuechecker.transactionHash
```

The output of this command is the transaction hash of the contract creation transaction.

```
"0x73fcceace856553513fa25d0ee9e4a085e23a508e17ae811960b0e28a198efab"
```

We can also query the ABI, using the following command:

```
> valuechecker.abi
```

The output will be as shown as follows. Note that it shows all the inputs and outputs of our example contract.

```
[{
    anonymous: false,
    inputs: [{
        indexed: false,
        internalType: "bool",
        name: "returnValue",
        type: "bool"
    }],
    name: "valueEvent",
    type: "event"
}, {
    inputs: [{
        internalType: "uint8",
        name: "x",
        type: "uint8"
    }],
    name: "Matcher",
    outputs: [{
        internalType: "bool",
        name: "",
        type: "bool"
    }],
    stateMutability: "nonpayable",
    type: "function"
}]
```

In the following example, the `Matcher` function is called with the arguments. Arguments, also called parameters, are the values passed to the functions. Remember that in the smart contract code shown in *Figure 15.2*, there is a condition that checks if the value is equal to or greater than 10, and if so, the function returns `true`; otherwise, it returns `false`. To test this, type the following commands into the `geth` console that you have open.

Pass `12` as an argument, which will return `true` as it is greater than 10:

```
> valuechecker.Matcher.call(12)
true
```

Pass `10` as an argument, which will return `true` as it is equal to 10:

```
> valuechecker.Matcher.call(10)
true
```

Pass `9` as an argument, which will return `false` as it is less than 10:

```
> valuechecker.Matcher.call(9)
false
```

In this section, we learned how to use the Remix IDE to create and deploy contracts. We also learned how the `geth` console can be used to interact with a smart contract and explored which methods are available to interact with smart contracts on the blockchain. Now we'll see how we can interact with `geth` using JSON RPC over HTTP.

POST requests

It is possible to interact with `geth` via JSON RPC over HTTP. For this purpose, the `curl` tool can be used.

`curl` is available at `https://curl.haxx.se/`.

An example is shown here to familiarize you with the POST request and show how to make POST requests using `curl`.

POST is a request method supported by HTTP. You can read more about POST here: `https://en.wikipedia.org/wiki/POST_(HTTP)`.

Before using the JSON RPC interface over HTTP, the `geth` client should be started up with appropriate switches, as shown here:

```
--rpcapi web3
```

This switch will enable the `web3` interface over HTTP. The Linux command, `curl`, can be used for the purpose of communicating over HTTP, as shown in the following example.

Retrieving the list of accounts

For example, in order to retrieve the list of accounts using the `personal_listAccounts` method, the following command can be used:

```
$ curl --request POST --data '{"jsonrpc":"2.0","method":"personal_
listAccounts","params": [],"id":4}' localhost:8545 -H "Content-Type: application/
json"
```

This will return the output, a JSON object with the list of accounts:

```
{"jsonrpc":"2.0","id":4,"result":["0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811","0
xd6e364a137e8f528ddbad2bb2356d124c9a08206"]}
```

In the preceding `curl` command, `--request` is used to specify the request command, `POST` is the request, and `--data` is used to specify the parameters and values. Finally, `localhost:8545` is used where the HTTP endpoint from Geth is opened.

In this section, we covered how we can interact with the smart contract using the JSON RPC over HTTP. While this is a common way of interacting with the contracts, the examples we have seen so far are command line-based. In the next section, we'll see how we can interact with the contracts by creating a user-friendly web interface.

Interacting with contracts via frontends

It is desirable to interact with the contracts in a user-friendly manner via a webpage. It is possible to interact with the contracts using the `web3.js` library from HTML-/JS-/CSS-based webpages.

The HTML and JavaScript frontend

The HTML content can be served using any HTTP web server, whereas `web3.js` can connect via local RPC to the running Ethereum client (`geth`) and provide an interface to the contracts on the blockchain. This architecture can be visualized in the following diagram:

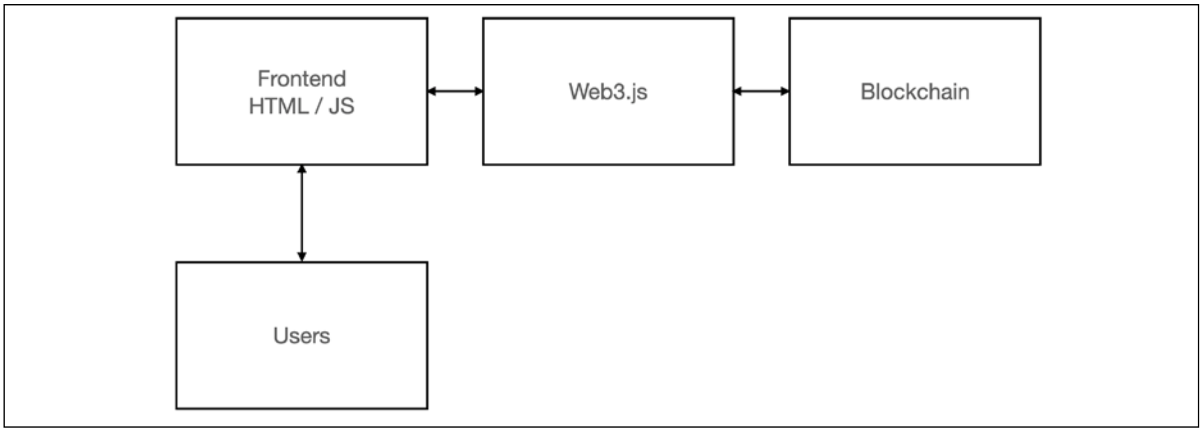


Figure 15.6: web3.js, frontend, and blockchain interaction architecture

If `web3.js` is not JavaScript frontend already installed, use these steps; otherwise, move on to the next section, *Interacting with contracts via a web frontend*.

Installing Web3.js JavaScript library

Web3, which we discussed earlier in this chapter, was looked at in the context of the Web3 API exposed by `geth`. In this section, we will introduce the Web3 JavaScript library (`web3.js`), which is used to introduce different functionalities related to the Ethereum ecosystem in DApps. The `web3.js` library is a collection of several modules, which are listed as follows with the functionality that they provide.

- `web3-eth`: Ethereum blockchain and smart contracts
- `web3-shh`: Whisper protocol (P2P communication and broadcast)
- `web3-bzz`: Swarm protocol, which provides decentralized storage
- `web3-utils`: Provides helper functions for DApp development

The `web3.js` library can be installed via `npm` by simply issuing the following command:

```
$ npm install web3
```

> `web3.js` can also be directly downloaded from `https://github.com/ethereum/web3.js`.

`web3.min.js`, downloaded via `npm`, can be referenced in the HTML files. This can be found under `node_modules`, for example, `/home/drequinox/netstats/node_modules/web3/dist/web3.js`.

> Note that `drequinox` is specific to the user under which these examples were developed; you will see the name of the user that you are running these commands under.

The file can optionally be copied into the directory where the main application is and can be used from there. Once the file is successfully referenced in HTML or JavaScript, Web3 needs to be initialized by providing an HTTP provider. This is usually the link to the `localhost` HTTP endpoint exposed by running the `geth` client. This can be achieved using the following code:

```
web3.setProvider(new web3.providers.HttpProvider('http://localhost:8545'));
```

Once the provider is set, further interaction with the contracts and blockchain can be done using the `web3` object and its available methods. The `web3` object can be created using the following code:

```
if (typeof web3 !== 'undefined')
{
    web3 = new Web3(web3.currentProvider);
}
else
{
    web3 = new Web3(new
    Web3.providers.HttpProvider("http://localhost:8545"));
}
```

In this section, we have explored how to install `web3.js`, the Ethereum JavaScript API library, and how to create a `web3` object that can be used to interact with the smart contracts using the HTTP provider running on the `localhost` as part of the Geth instance.

Interacting with contracts via a web frontend

In the following section, an example will be presented that will make use of `web3.js` to allow interaction with the contracts, via a webpage served over a simple HTTP web server. So far, we have seen how we can interact with a contract using the `geth` console via the command line, but in order for an application to be usable by end users, who will mostly be familiar with web interfaces, it becomes necessary to build web frontends so that users can communicate with the backend smart contracts using familiar webpage interfaces.

Creating an app.js JavaScript file

This can be achieved by following these steps. First, create a directory named `/simplecontract/app`, the home directory. This is the main directory under your user ID on Linux or macOS. This can be any directory, but in this example, the home directory is used.

Then, create a file named `app.js`, and write or copy the following code into it:

```
var Web3 = require('web3');

if (typeof web3 !== 'undefined') {
  web3 = new Web3(web3.currentProvider);
} else {
  // set the provider you want from Web3.providers
  web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
}

web3.eth.defaultAccount = web3.eth.accounts[0];

var SimpleContract = web3.eth.contract([
    {
        "constant": false,
        "inputs": [
            {
                "name": "x",
                "type": "uint8"
            }
        ],
        "name": "Matcher",
        "outputs": [
            {
                "name": "",
                "type": "bool"
            }
        ],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": false,
                "name": "returnValue",
                "type": "bool"
            }
        ],
        "name": "valueEvent",
        "type": "event"
    }
]);

var simplecontract = SimpleContract.at("0x82012b7601fd23669b50bb7dd79460970ce38
6e3");
        console.log(simplecontract);


function callMatchertrue()
{
var txn = simplecontract.Matcher.call(12);
{
};

console.log("return value: " + txn);
}

function callMatcherfalse()
{
var txn = simplecontract.Matcher.call(1);{
};
console.log("return value: " + txn);
}

function myFunction()
{
  var x = document.getElementById("txtValue").value;

    var txn = simplecontract.Matcher.call(x);{
};
console.log("return value: " + txn);
```

```
    document.getElementById("decision").innerHTML = txn;
}
```

This file contains various elements; first, we created the `web3` object and provided a `localhost` `geth` instance listening on port `8545` as the Web3 provider. After this, the `web3.eth.accounts[0]` is selected as the account with which all the interactions will be performed with the smart contract. Next, the ABI is provided, which serves as the interface between the user and the contract. It can be queried using `geth`, generated using the Solidity compiler, or copied directly from the Remix IDE contract details. After this, the `simplecontract` is created, which refers to the smart contract with address `0x82012b7601fd23669b50bb7dd79460970ce386e3`. Finally, we declared three functions: `callMatchertrue()`, `callMatcherfalse()`, and `myFunction()`, which we will explain further shortly.

First, we create the frontend webpage. For this, we create a file named `index.html` with the source code shown as follows:

```
<html>
<head>
<title>SimpleContract Interactor</title>
<script src="./web3.js"></script>
<script src="./app.js"></script>
</head>

<body>

<p>Enter your value:</p>
<input type="text" id="txtValue" value="">

<p>Click the "get decision" button to get the decision from the smart contract.</
p>

<button onclick="myFunction()">get decision</button>

<p id="decision"></p>

<p>Calling the contract manually with hardcoded values, result logged in browser
debug console:</p>
<button onclick="callMatchertrue()">callTrue</button>
<button onclick="callMatcherfalse()">callFalse</button>

</body>
</html>
```

This file will serve as the frontend of our decentralized application. In other words, it provides the UI for interacting with the smart contracts. First, we referred to the JavaScript `web3.js` library and `app.js`, which we created earlier in this section. This will allow the HTML file to call required functions from these files.

After that, standard HTML is used to create an input text field so that users can enter the values. Then we used the `onclick` event to call the `myFunction()` function that we declared in our JavaScript `app.js` file. Finally, two `onclick` events with buttons `callTrue` and `callFalse` are used to call the `callMatchertrue()` and `callMatcherfalse()` functions, respectively.

We are keeping this very simple on purpose; there is no direct need to use jQuery, React, or Angular here, which would be a separate topic. Nevertheless, these frontend frameworks make development easier and a lot faster, and are commonly used for blockchain-related JavaScript frontend development.

In order to keep things simple, we are not going to use any frontend JavaScript frameworks in this section, as the main aim is to focus on blockchain technology and not the HTML, CSS, or JavaScript UI frameworks. However, in the bonus resource pack for this chapter, we will see an example of creating UIs using `react` with Drizzle.

In this part of the example, we have created a web frontend and a JavaScript file backend, where we have defined all the functions required for our application.

The `app.js` file we created in this section is the main JavaScript file that contains the code to create a `web3` object. It also provides methods that are used to interact with the contract on the blockchain. An explanation of the code previously used is given in the next sections.

Creating a Web3 object

The first step when creating a `web3.js`-based application is to create the `web3` object. It is created by selecting the appropriate available Web3 provider, which serves as an "entry point" to the blockchain through the HTTP RPC server exposed on a locally running Geth node.

```
if (typeof web3 !== 'undefined')
{
web3 = new Web3(web3.currentProvider);
}
else
{
web3 = new Web3(new Web3.providers.HttpProvider("http://localhost: 8545"));
}
```

This code checks whether there is already an available Web3 provider; if yes, then it will set the provider to the current provider. Otherwise, it sets the provider to `localhost: 8545`; this is where the local instance of `geth` is exposing the HTTP-RPC server. In other words, the `geth` instance is running an HTTP-RPC server, which is listening on port `8545`.

Checking availability by calling the Web3 method

This line of code simply uses `console.log` to print the simple contract attributes, in order to verify the successful creation of the contract:

```
var simplecontract = SimpleContract.at("0x82012b7601fd23669b50bb7dd79460970ce38
6e3");
console.log(simplecontract);
```

Once this call is executed, it will display various contract attributes indicating that the `web3` object has been created successfully and `HttpProvider` is available. Any other call can be used to verify the availability, but here, printing simple contract attributes has been used.

In this part of the example, we have created a `simplecontract` instance and then used `console.log` to display some attributes of the contract, which indicates the successful creation of the `simplecontract` instance. In the next stage of this example, we will explore how the contract functions can be called.

Calling contract functions

Once the `web3` object is correctly created and a `simplecontract` instance is created (in the previous section), calls to the contract functions can be made easily as shown in the following code:

```
function callMatchertrue()
{
var txn = simplecontract.Matcher.call(12);
{
};

console.log("return value: " + txn);
}

function callMatcherfalse()
{
var txn = simplecontract.Matcher.call(1);{
};
console.log("return value: " + txn);
}

function myFunction()
{
  var x = document.getElementById("txtValue").value;

    var txn = simplecontract.Matcher.call(x);{
};
console.log("return value: " + txn);
    document.getElementById("decision").innerHTML = txn;
}
```

The preceding code shows three simple functions, `callMatchertrue()`, `callMatcherfalse()`, and `myFunction()`. `callMatchertrue()` simply calls the smart contract function matcher using the `simplecontract` object we created in the `app.js` file we created earlier. Similarly, `callMatcherfalse()` calls the smart contract's `Matcher` function by providing a value of `1`. Finally, the `myFunction()` function is defined, which contains simple logic to read the value provided by the user on the webpage in a `txtValue` textbox and uses that value to call the smart contract's matcher function.

After that, the return value is also logged in the debug console, available in browsers by using `console.log`.

Calls can be made using `simplecontractinstance.Matcher.call` and then by passing the value for the argument. Recall the `Matcher` function in the Solidity code:

```
function Matcher (uint8 x) returns (bool)
```

It takes one argument `x` of type `uint8` and returns a Boolean value, either `true` or `false`. Accordingly, the call is made to the contract, as shown here:

```
var txn = simplecontractinstance.Matcher.call(12);
```

In the preceding example, `console.log` is used to print the value returned by the function call. Once the result of the call is available in the `txn` variable, it can be used anywhere throughout the program, for example, as a parameter for another JavaScript function.

Finally, the HTML file named `index.html` is created with the following code. This HTML file will serve as the frontend UI for the users, who can browse to this page served via an HTTP server to interact with the smart contract:

```
<html>
<head>
<title>SimpleContract Interactor</title>
<script src="./web3.js"></script>
<script src="./app.js"></script>
</head>

<body>

<p>Enter your value:</p>
<input type="text" id="txtValue" value="">

<p>Click the "get decision" button to get the decision from the smart contract.</
p>

<button onclick="myFunction()">get decision</button>

<p id="decision"></p>
```

```
<p>Calling the contract manually with hardcoded values, result logged in browser
debug console:</p>
<button onclick="callMatchertrue()">callTrue</button>
<button onclick="callMatcherfalse()">callFalse</button>

</body>
</html>
```

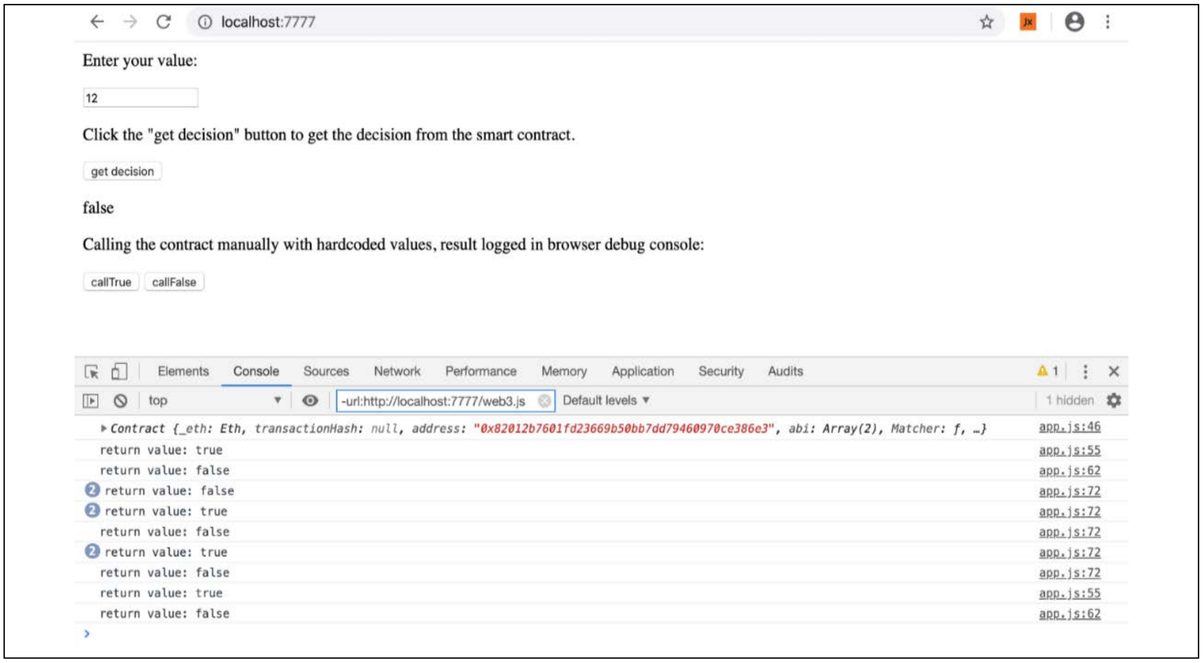
It is recommended that a web server is running in order to serve the HTML content (`index.html` as an example). Alternatively, the file can be browsed from the filesystem but that can cause some issues related to serving the content correctly with larger projects; as a good practice, always use a web server.

A web server in Python can be started using the following command. This server will serve the HTML content from the same directory that it has been run from:

```
$ python -m SimpleHTTPServer 7777
Serving HTTP on 0.0.0.0 port 7777 ...
```

The web server does not have to be in Python; it can be an Apache server or any other web container.

Now any browser can be used to view the webpage served over TCP port `7777`. This is shown in the following screenshot:



Enter your value:

12

Click the "get decision" button to get the decision from the smart contract.

false

Calling the contract manually with hardcoded values, result logged in browser debug console:

```
▸Contract {_eth: Eth, transactionHash: null, address: "0x82012b7601fd23669b50bb7dd79460970ce386e3", abi: Array(2), Matcher: f, …}   app.js:46
  return value: true                                                                                                             app.js:55
  return value: false                                                                                                            app.js:62
2 return value: false                                                                                                            app.js:72
2 return value: true                                                                                                             app.js:72
  return value: false                                                                                                            app.js:72
2 return value: true                                                                                                             app.js:72
  return value: false                                                                                                            app.js:72
  return value: true                                                                                                             app.js:55
  return value: false                                                                                                            app.js:62
>
```

Figure 15.7: Interaction with the contract

It should be noted that the output shown here is in the browser's console window. The browser's console must be enabled in order to see the output. For example, in Chrome you can use keyboard shortcuts to open the console. On Windows and Linux, *Ctrl* + *Shift* + *J*, and on Mac, *Cmd* + *Option* + *J* are used.

As the values are hardcoded in the code for simplicity, two buttons shown in the screenshot, **callTrue** and **callFalse**, have been created in `index.html`. Both of these buttons call functions with hardcoded values. This is just to demonstrate that parameters are being passed to the contract via Web3 and values are being returned accordingly.

There are three functions being called behind these buttons. We will describe them as follows:

1. The `get decision` button returns the decision from the contract:

   ```
   <button onclick="myFunction()">get decision</button>
   function myFunction()
   {
     var x = document.getElementById("txtValue").value;

       var txn = simplecontract.Matcher.call(x);{
   };
   console.log("return value: " + txn);
       document.getElementById("decision").innerHTML = txn;
   }
   ```

   The `get decision` button invokes the smart contract's `Matcher` function with the value entered on the webpage. The variable `x` contains the value passed to this function via the webpage, which is `12`. As the value is `12`, which is greater than `10`, the `get decision` button will return `true`.

2. The `callTrue` button will call the `Matcher` function with a value that is always greater than 10, such as 12, returning always `true`. The `callMatchertrue()` method has a hardcoded value of `12`, which is sent to the contract using the following code:

   ```
   simplecontractinstance.Matcher.call(12)
   ```

   The return value is printed in the console using the following code, which first invokes the `Matcher` function and then assigns the value to the `txn` variable to be printed later in the console:

   ```
   simplecontractinstance.Matcher.call(1) function callMatchertrue()
   {
   var txn = simplecontractinstance.Matcher.call(12);{
   };
   console.log("return value: " + txn);
   }
   ```

3. The `callFalse` button: invokes the `callMatcherfalse()` function. The `callMatcherfalse()` function works by passing a hardcoded value of `1` to the contract using this code:

   ```
   simplecontractinstance.Matcher.call(1)
   ```

   The return value is printed accordingly:

   ```
   console.log("return value: " + txn);
   function callMatcherfalse()
   {
   var txn = simplecontractinstance.Matcher.call(1);{
   };
   console.log("return value: " + txn);
   }
   ```

Note that there is no real need for the `callTrue` and `callFalse` methods here; they are just presented for pedagogical reasons so that readers can correlate the functions with the hardcoded values and then to the called function within the smart contract, with `value` as a parameter.

This example demonstrates how the Web3 library can be used to interact with the contracts on the Ethereum blockchain. First, we created a web frontend using the JavaScript `app.js` file and the HTML file. We also included the Web3 library in our HTML so that we could create the `web3` object and use that to interact with the deployed smart contract.

In the next section, we will explore some development frameworks that aid Ethereum development, including a commonly used framework called Truffle.

Development frameworks

There are various development frameworks now available for Ethereum. As seen in the examples discussed earlier, it can be quite time-consuming to deploy the contracts via manual means. This is where **Truffle** and similar frameworks such as Embark can be used to make the process simpler and quicker. We have chosen Truffle because it has a more active developer community and is currently the most widely used framework for Ethereum development. However, note that there is no best framework as all frameworks aim to provide methods to make development, testing, and deployment easier.

You can read more about Embark here: `https://github.com/embark-framework/embark`.

In the next section, you will be introduced to an example project to demonstrate the usage of the Truffle framework.

Using Truffle to develop a decentralized application

We discussed Truffle briefly in *Chapter 14*, *Development Tools and Frameworks*. In this section, we will see an example project that will demonstrate how Truffle can be used to develop a decentralized application. We will see all the steps involved in this process such as initialization, testing, migration, and deployment. First, we will see the installation process.

Installing and initializing Truffle

If Truffle is not already installed, it can be installed by running the following command:

```
$ npm install -g truffle
```

Next, Truffle can be initialized by running the following commands. First, create a directory for the project, for example:

```
$ mkdir testdapp
```

Then, change the directory to the newly created `testdapp` and run the following command:

```
$ truffle init
✓ Preparing to download box
✓ Downloading
✓ cleaning up temporary files
✓ Setting up box
```

Once the command is successful, it will create the directory structure, as shown here. This can be viewed using the `tree` command in Linux:

```
$ tree
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
└── truffle-config.js

3 directories, 3 files
```

This command creates three main directories named `contracts`, `migrations`, and `test`. As seen in the preceding example, a total of 3 directories and 4 files have been created. The directories are defined here:

- `contracts`: This directory contains Solidity contract source code files. This is where Truffle will look for Solidity contract files during migration.
- `migration`: This directory has all the deployment scripts.

- `test`: As the name suggests, this directory contains relevant test files for applications and contracts.

Finally, Truffle configuration is stored in the `truffle.js` file, which is created in the root folder of the project from where `truffle init` was run.

When `truffle init` is run, it will create a skeleton tree with files and directories. In previous versions of Truffle, this used to produce a project named **MetaCoin** but now, this project is now available as a **Truffle box**. Now that we have initialized Truffle, let's see how it is used to compile, test, and migrate smart contracts.

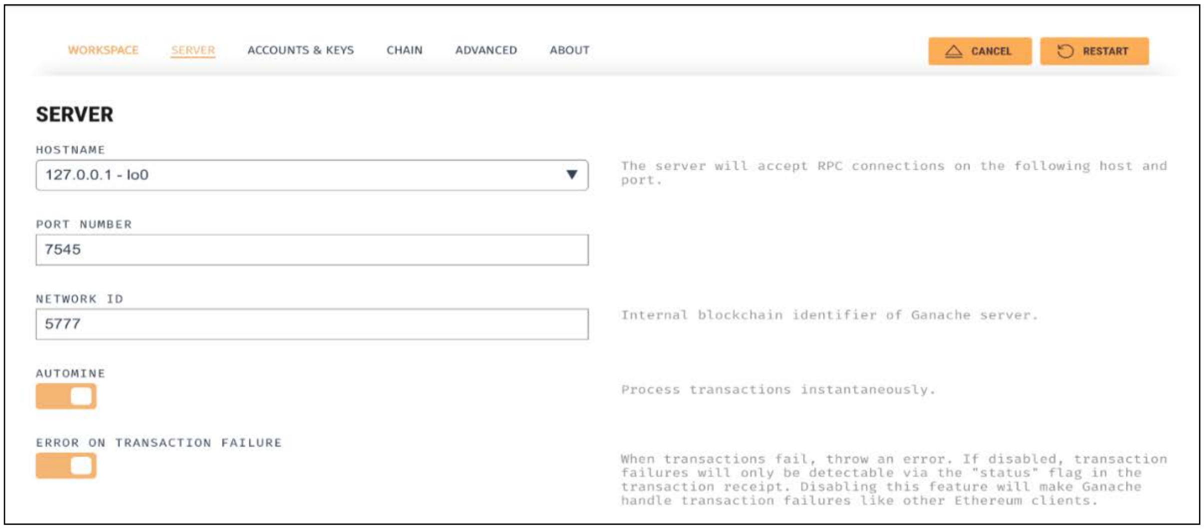
Compiling, testing, and migrating using Truffle

In this section, we will demonstrate how to use various operations available in Truffle. We will introduce how to use compilation, testing, and migration commands in Truffle to deploy and test **Truffle boxes**, which are essentially sample projects available with Truffle. We will use the MetaCoin Truffle box. Later, further examples will be shown on how to use Truffle for custom projects.

We will use Ganache as a local blockchain to provide the RPC interface. Make sure that Ganache is running in the background and mining.

We covered Ganache setup in *Chapter 14*, *Development Tools and Frameworks*. You can refer to this chapter for a refresher.

In the following example, Ganache is running on port `7545` with 10 accounts. These options can be changed in the **Settings** option in Ganache as shown in the following screenshot:



SERVER

HOSTNAME
127.0.0.1 - lo0
The server will accept RPC connections on the following host and port.

PORT NUMBER
7545

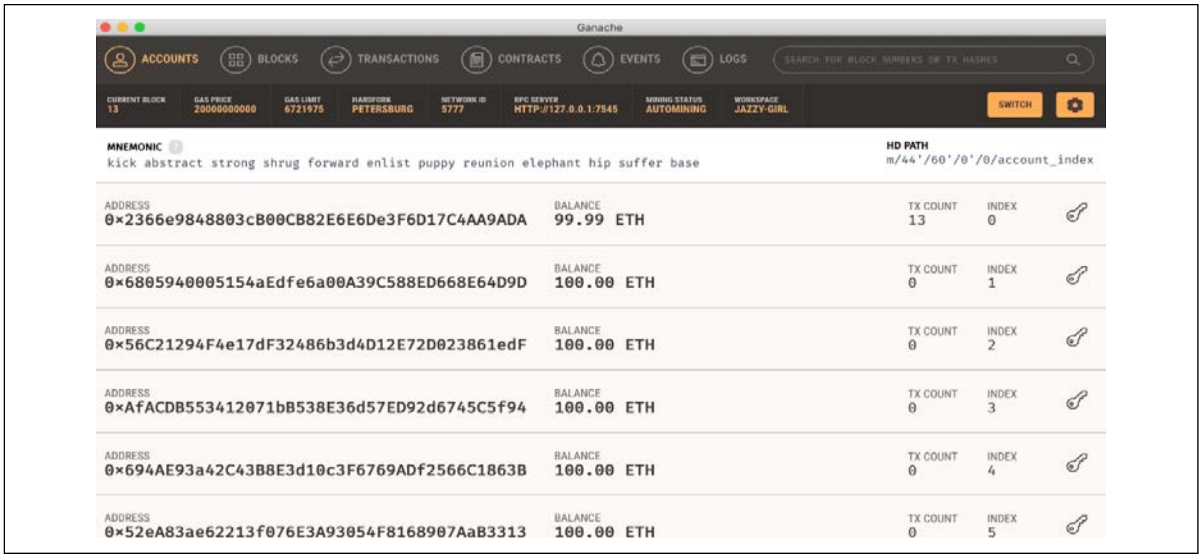
NETWORK ID
5777
Internal blockchain identifier of Ganache server.

AUTOMINE
Process transactions instantaneously.

ERROR ON TRANSACTION FAILURE
When transactions fail, throw an error. If disabled, transaction failures will only be detectable via the "status" flag in the transaction receipt. Disabling this feature will make Ganache handle transaction failures like other Ethereum clients.

Figure 15.8: Ganache settings

We will use the Ganache workspace that we saved in *Chapter 14*, *Development Tools and Frameworks*. Alternatively, a new environment can also be set up.



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
13	20000000000	6721975	PETERSBURG	5777	HTTP://127.0.0.1:7545	AUTOMINING	JAZZY-GIRL

MNEMONIC
kick abstract strong shrug forward enlist puppy reunion elephant hip suffer base

HD PATH
m/44'/60'/0'/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA	99.99 ETH	13	0
0×6805940005154aEdfe6a00A39C588ED668E64D9D	100.00 ETH	0	1
0×56C21294F4e17dF32486b3d4D12E72D023861edF	100.00 ETH	0	2
0×AfACDB553412071bB538E36d57ED92d6745C5f94	100.00 ETH	0	3
0×694AE93a42C43B8E3d10c3F6769ADf2566C1863B	100.00 ETH	0	4
0×52eA83ae62213f076E3A93054F8168907AaB3313	100.00 ETH	0	5

Figure 15.9: Ganache view

After the successful setup of Ganache, the following steps need to be performed in order to unpack the webpack Truffle box and run the MetaCoin project. This example provides a solid foundation for the upcoming sections. With this exercise, we will learn how a sample project available with Truffle can be downloaded, and how we perform compilation, testing, and migration of the contracts available with this sample onto Ganache:

1. First, create a new directory:

   ```
   $ mkdir tproject
   $ cd tproject
   ```

2. Now, unbox the webpack sample from Truffle:

   ```
   $ truffle unbox metacoin
   ✓ Preparing to download box
   ✓ Downloading
   ✓ cleaning up temporary files
   ✓ Setting up box
   ```

3. Edit the `truffle.js` file: if Ganache is running on a different port then change the port from the default to where Ganache is listening. Note the settings provided in the screenshot in *Figure 15.8*:

   ```
   $ cat truffle-config.js
   module.exports = {
     // Uncommenting the defaults below
     // provides for an easier quick-start with Ganache.
     // You can also follow this format for other networks;
   ```

```
  // see <http://truffleframework.com/docs/advanced/configuration>
  // for more details on how to specify configuration options!
  //
  //networks: {
  //  development: {
  //    host: "127.0.0.1",
  //    port: 7545,
  //    network_id: "*"
  //  },
  //  test: {
  //    host: "127.0.0.1",
  //    port: 7545,
  //    network_id: "*"
  //  }
  //}
  //
};
```

Edit the file and uncomment the defaults. The file should look like the one shown here:

```
module.exports = {
  // Uncommenting the defaults below
  // provides for an easier quick-start with Ganache.
  // You can also follow this format for other networks;
  // see <http://truffleframework.com/docs/advanced/configuration>
  // for more details on how to specify configuration options!
  //
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    },
    test: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    }
  }
};
```

Now, after unboxing the webpack sample and making the necessary configuration changes, we are ready to compile all the contracts.

4. Now run the following command to compile all the contracts:

   ```
   $ truffle compile
   ```

   This will show the following output:

   ```
   Compiling your contracts...
   ===========================
   > Compiling ./contracts/ConvertLib.sol
   > Compiling ./contracts/MetaCoin.sol
   > Compiling ./contracts/Migrations.sol
   > Artifacts written to /Users/drequinox/tproject/build/contracts
   > Compiled successfully using:
      - solc: 0.5.16+commit.9c3226ce.Emscripten.clang
   ```

5. Now we can test using the `truffle test` command as shown here:

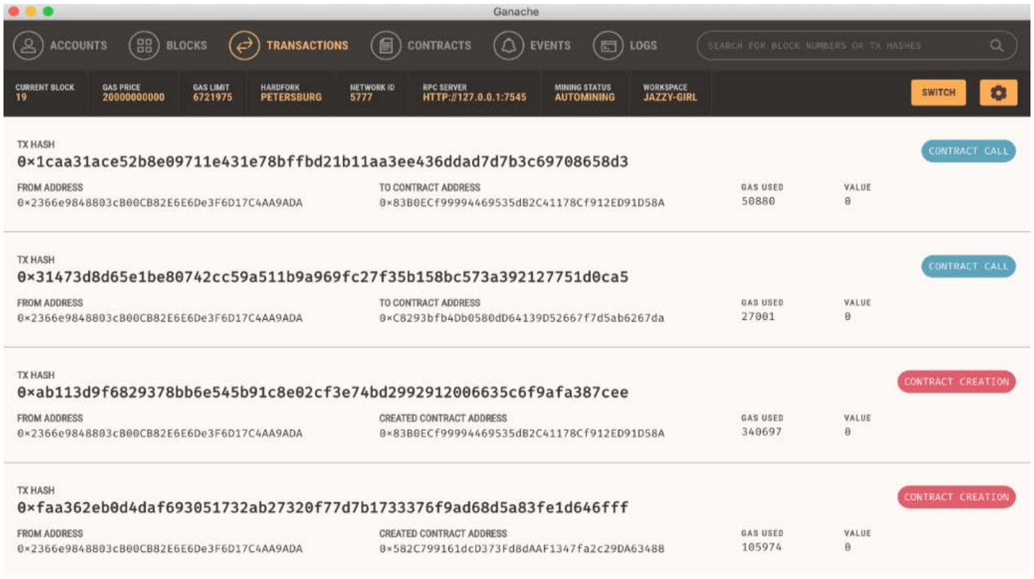
   ```
   $ truffle test
   ```

   This command will produce the output shown as follows, indicating the progress of the testing process:

   ```
   Using network 'development'.

   Compiling your contracts...
   ===========================
   > Compiling ./test/TestMetaCoin.sol
   > Artifacts written to /var/folders/82/5r_y_y_13wq4nqb0fw6s52nc0000gn
   /T/test-202017-5530-1piqoce.3i7ch
   > Compiled successfully using:
      - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

   TruffleConfig {
     _deepCopy: [ 'compilers' ],
     _values:
   .
   .  . . . . . . . . text not shown for brevity . . . . . . . . . .
   .
   test_files:
      [ '/Users/drequinox/tproject/test/metacoin.js',
        '/Users/drequinox/tproject/test/TestMetaCoin.sol' ] }


     TestMetaCoin
       ✓ testInitialBalanceUsingDeployedContract (122ms)
       ✓ testInitialBalanceWithNewMetaCoin (186ms)

     Contract: MetaCoin
   ```

```
      ✓ should put 10000 MetaCoin in the first account
      ✓ should call a function that depends on a linked library (120ms)
      ✓ should send coin correctly (182ms)
```

We can also see that in Ganache, the transactions are being processed as a result of running the test:



CURRENT BLOCK 19 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK PETERSBURG | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING | WORKSPACE JAZZY-GIRL

TX HASH
0×1caa31ace52b8e09711e431e78bffbd21b11aa3ee436ddad7d7b3c69708658d3 — CONTRACT CALL
FROM ADDRESS 0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA
TO CONTRACT ADDRESS 0×83B0ECf99994469535dB2C41178Cf912ED91D58A
GAS USED 50880
VALUE 0

TX HASH
0×31473d8d65e1be80742cc59a511b9a969fc27f35b158bc573a392127751d0ca5 — CONTRACT CALL
FROM ADDRESS 0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA
TO CONTRACT ADDRESS 0×C8293bfb4Db0580dD64139D52667f7d5ab6267da
GAS USED 27001
VALUE 0

TX HASH
0×ab113d9f6829378bb6e545b91c8e02cf3e74bd2992912006635c6f9afa387cee — CONTRACT CREATION
FROM ADDRESS 0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA
CREATED CONTRACT ADDRESS 0×83B0ECf99994469535dB2C41178Cf912ED91D58A
GAS USED 340697
VALUE 0

TX HASH
0×faa362eb0d4daf693051732ab27320f77d7b1733376f9ad68d5a83fe1d646fff — CONTRACT CREATION
FROM ADDRESS 0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA
CREATED CONTRACT ADDRESS 0×582C799161dcD373Fd8dAAF1347fa2c29DA63488
GAS USED 105974
VALUE 0

Figure 15.10: Truffle screen as a result of testing

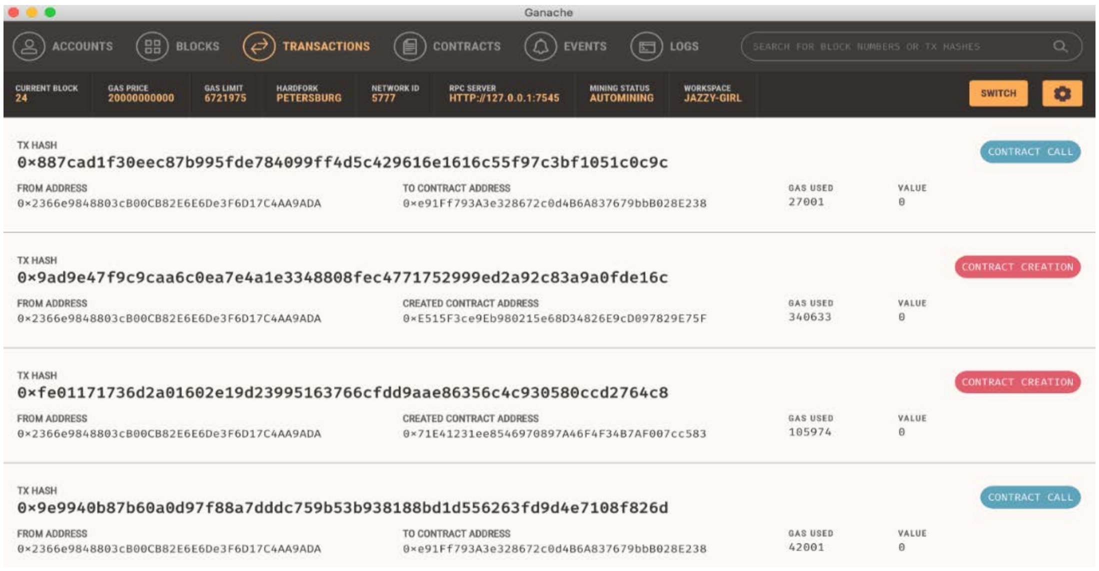
6. Once testing is completed, we can migrate to the blockchain using the following command. Migration will use the settings available in `truffle.js` that we edited in the second step of this process to point to the port where Ganache is running. This is achieved by issuing the following command:

```
$ truffle migrate
```

The output is shown as follows. Notice that when migration runs it will reflect on Ganache; for example, the balance of accounts will go down and you can also view transactions that have been executed. Also notice that the account shown in the screenshot corresponds with what is shown in Ganache:

```
Starting migrations...
======================
> Network name:    'development'
> Network id:      5777
Block gas limit: 0x6691b7


1_initial_migration.js
======================
Deploying 'Migrations'
----------------------
transaction hash:
0x47ce80861f3965036a4c6b78cb4cc03b3a77162ebeebaf851859fe59e16aff1e
> Blocks: 0            Seconds: 0
> contract address:    0xe91Ff793A3e328672c0d4B6A837679bbB028E238
> block number:        20
> block timestamp:     1581113420
> account:             0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA
> balance:             99.97009608
> gas used:            188483
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.00376966 ETH


Saving migration to chain.
Saving artifacts
-------------------------------------
> Total cost:          0.00376966 ETH


2_deploy_contracts.js
=====================

Deploying 'ConvertLib'
----------------------
transaction hash:
0xfe01171736d2a01602e19d23995163766cfdd9aae86356c4c930580ccd2764c8
> Blocks: 0            Seconds: 0
> contract address:    0x71E41231ee8546970897A46F4F34B7AF007cc583
> block number:        22
> block timestamp:     1581113421
> account:             0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA
> balance:             99.96713658
> gas used:            105974
> gas price:            20 gwei
> value sent:           0 ETH
> total cost:           0.00211948 ETH


Linking
-------
Contract: MetaCoin <--> Library: ConvertLib (at address:
0x71E41231ee8546970897A46F4F34B7AF007cc583)

Deploying 'MetaCoin'
--------------------
transaction hash:
0x9ad9e47f9c9caa6c0ea7e4a1e3348808fec4771752999ed2a92c83a9a0fde16c
> Blocks: 0             Seconds: 0
> contract address:     0xE515F3ce9Eb980215e68D34826E9cD097829E75F
> block number:         23
> block timestamp:      1581113421
> account:              0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA
> balance:              99.96032392
> gas used:             340633
> gas price:            20 gwei
> value sent:           0 ETH
> total cost:           0.00681266 ETH


Saving migration to chain.
Saving artifacts
-------------------------------------
> Total cost:           0.00893214 ETH


Summary
=======
> Total deployments:   3
> Final cost:           0.0127018 ETH
```

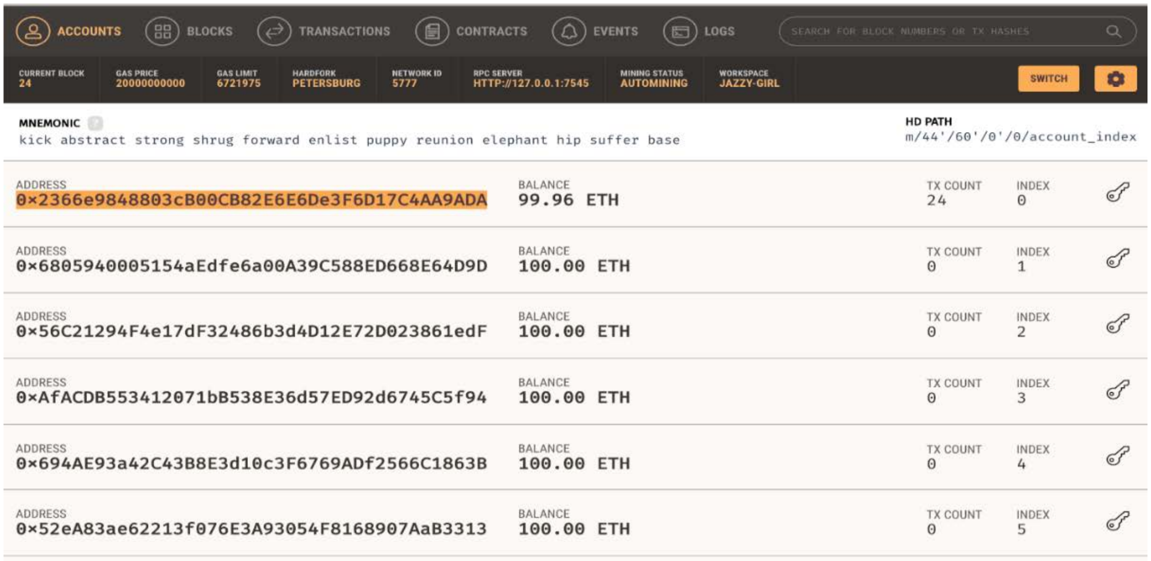
Notice that the migration we have just performed is reflected in Ganache with the accounts shown previously:



```
CURRENT BLOCK  GAS PRICE    GAS LIMIT  HARDFORK    NETWORK ID  RPC SERVER              MINING STATUS  WORKSPACE
24             20000000000  6721975    PETERSBURG  5777        HTTP://127.0.0.1:7545   AUTOMINING     JAZZY-GIRL

TX HASH
0×887cad1f30eec87b995fde784099ff4d5c429616e1616c55f97c3bf1051c0c9c                                      CONTRACT CALL
FROM ADDRESS                                 TO CONTRACT ADDRESS                          GAS USED   VALUE
0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA   0×e91Ff793A3e328672c0d4B6A837679bbB028E238   27001      0

TX HASH
0×9ad9e47f9c9caa6c0ea7e4a1e3348808fec4771752999ed2a92c83a9a0fde16c                                      CONTRACT CREATION
FROM ADDRESS                                 CREATED CONTRACT ADDRESS                     GAS USED   VALUE
0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA   0×E515F3ce9Eb980215e68D34826E9cD097829E75F   340633     0

TX HASH
0×fe01171736d2a01602e19d23995163766cfdd9aae86356c4c930580ccd2764c8                                      CONTRACT CREATION
FROM ADDRESS                                 CREATED CONTRACT ADDRESS                     GAS USED   VALUE
0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA   0×71E41231ee8546970897A46F4F34B7AF007cc583   105974     0

TX HASH
0×9e9940b87b60a0d97f88a7dddc759b53b938188bd1d556263fd9d4e7108f826d                                      CONTRACT CALL
FROM ADDRESS                                 TO CONTRACT ADDRESS                          GAS USED   VALUE
0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA   0×e91Ff793A3e328672c0d4B6A837679bbB028E238   42001      0
```

Figure 15.11: Ganache displaying transactions

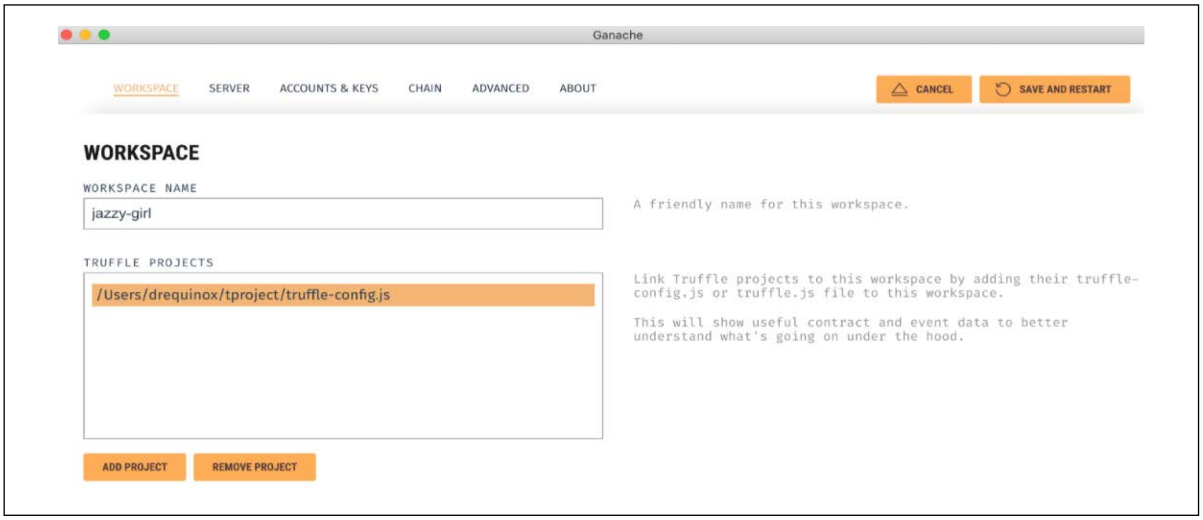
You can see **BALANCE** updating in Ganache, as the transactions run and ETH is consumed:



```
CURRENT BLOCK  GAS PRICE    GAS LIMIT  HARDFORK    NETWORK ID  RPC SERVER              MINING STATUS  WORKSPACE
24             20000000000  6721975    PETERSBURG  5777        HTTP://127.0.0.1:7545   AUTOMINING     JAZZY-GIRL

MNEMONIC                                                                       HD PATH
kick abstract strong shrug forward enlist puppy reunion elephant hip suffer base    m/44'/60'/0'/0/account_index

ADDRESS                                      BALANCE      TX COUNT  INDEX
0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA   99.96 ETH    24        0

ADDRESS                                      BALANCE      TX COUNT  INDEX
0×6805940005154aEdfe6a00A39C588ED668E64D9D   100.00 ETH   0         1

ADDRESS                                      BALANCE      TX COUNT  INDEX
0×56C21294F4e17dF32486b3d4D12E72D023861edF   100.00 ETH   0         2

ADDRESS                                      BALANCE      TX COUNT  INDEX
0×AfACDB553412071bB538E36d57ED92d6745C5f94   100.00 ETH   0         3

ADDRESS                                      BALANCE      TX COUNT  INDEX
0×694AE93a42C43B8E3d10c3F6769ADf2566C1863B   100.00 ETH   0         4

ADDRESS                                      BALANCE      TX COUNT  INDEX
0×52eA83ae62213f076E3A93054F8168907AaB3313   100.00 ETH   0         5
```

Figure 15.12: Ganache displaying accounts

In the Ganache workspace, we can also add the Truffle project to enable extra features. For this, click on the upper right-hand corner gear icon, and open the settings screen. Add the Truffle project as shown here:



WORKSPACE

WORKSPACE NAME

jazzy-girl

A friendly name for this workspace.

TRUFFLE PROJECTS

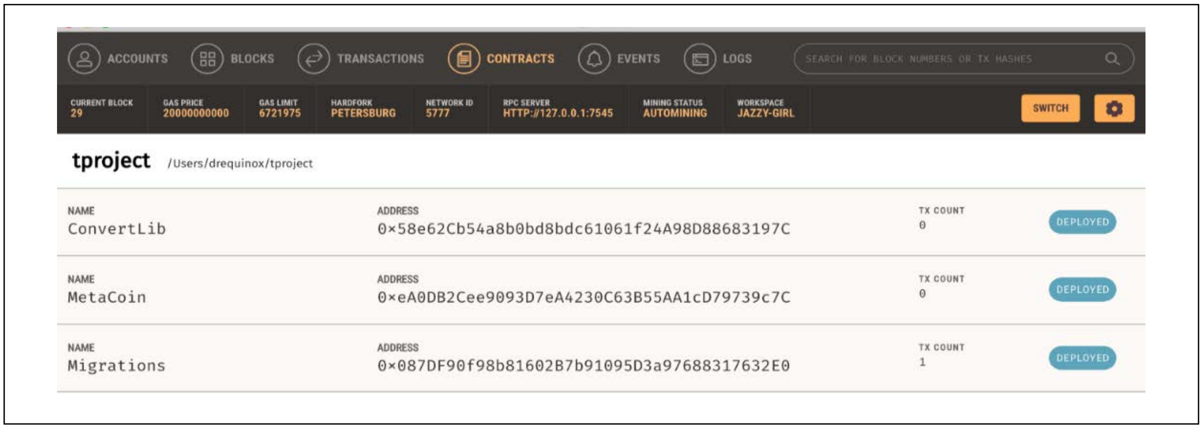
/Users/drequinox/tproject/truffle-config.js

Link Truffle projects to this workspace by adding their truffle-config.js or truffle.js file to this workspace.

This will show useful contract and event data to better understand what's going on under the hood.

Figure 15.13: Adding the Truffle project to Ganache

Save and restart to commit the changes. When Ganache is back up again, you will be able to see additional information about the contract as shown in the following screenshot:



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
29	20000000000	6721975	PETERSBURG	5777	HTTP://127.0.0.1:7545	AUTOMINING	JAZZY-GIRL

tproject /Users/drequinox/tproject

NAME	ADDRESS	TX COUNT	
ConvertLib	0×58e62Cb54a8b0bd8bdc61061f24A98D88683197C	0	DEPLOYED
MetaCoin	0×eA0DB2Cee9093D7eA4230C63B55AA1cD79739c7C	0	DEPLOYED
Migrations	0×087DF90f98b81602B7b91095D3a97688317632E0	1	DEPLOYED

Figure 15.14: More contract details are visible after adding the Truffle project

In this section so far, we've explored how Truffle can be used to compile, test, and deploy smart contracts on the blockchain. We used Ganache, the Ethereum personal blockchain (a simulated version of the Ethereum blockchain), to perform all the exercises.

Note that you will see slightly different outputs and screen depending on your local environment and Ganache and Truffle versions.

Now, we can interact with the contract using the Truffle console. We will explore this in the following section.

Interacting with the contract

Truffle also provides a console (a CLI) that allows interaction with the contracts. All deployed contracts are already instantiated and ready to use in the console. This is an **REPL**-based interface, meaning **Read**, **Evaluate**, and **Print Loop**. Similarly, in the `geth` client (via `attach` or `console`), REPL is used via exposing the **JavaScript runtime environment** (**JSRE**).

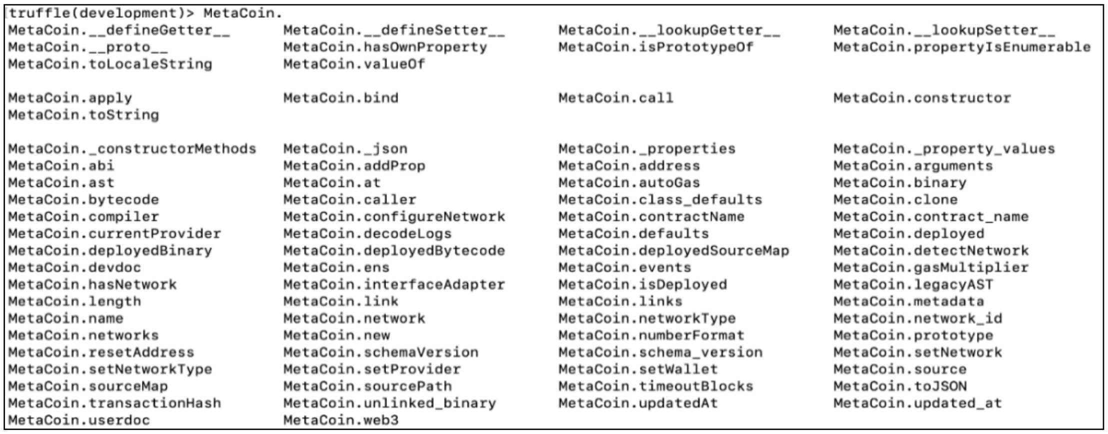
1. The console can be accessed by issuing the following command:

   ```
   $ truffle console
   ```

2. This will open a CLI:

   ```
   truffle(development)>
   ```

   Once the console is available, various methods can be run in order to query the contract. A list of methods can be displayed by typing the preceding command and tab-completing:



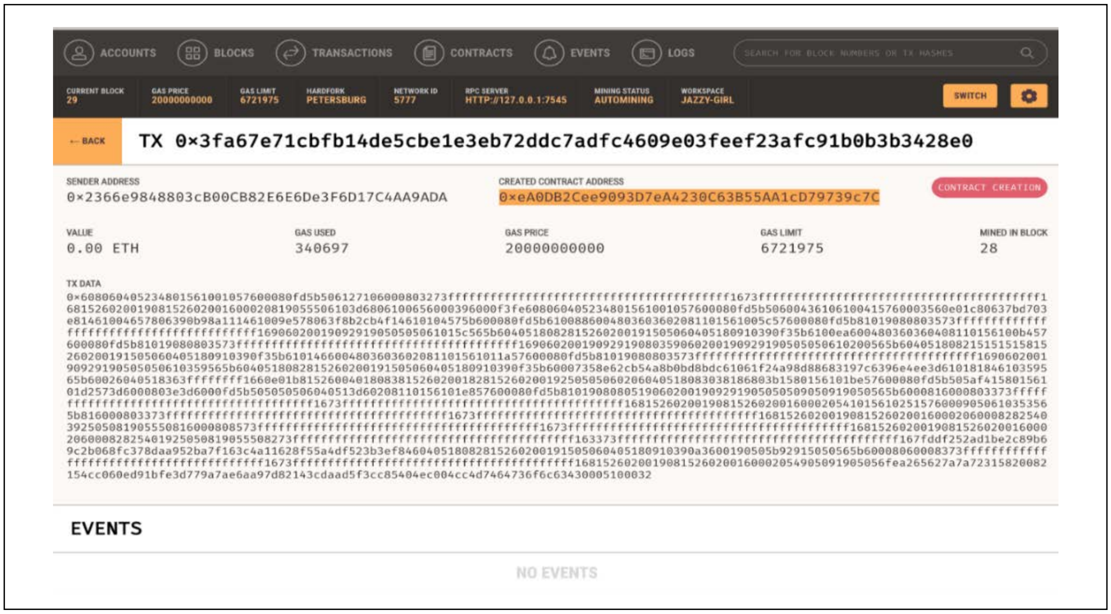
```
truffle(development)> MetaCoin.
MetaCoin.__defineGetter__       MetaCoin.__defineSetter__       MetaCoin.__lookupGetter__       MetaCoin.__lookupSetter__
MetaCoin.__proto__              MetaCoin.hasOwnProperty         MetaCoin.isPrototypeOf          MetaCoin.propertyIsEnumerable
MetaCoin.toLocaleString         MetaCoin.valueOf

MetaCoin.apply                  MetaCoin.bind                   MetaCoin.call                   MetaCoin.constructor
MetaCoin.toString

MetaCoin._constructorMethods    MetaCoin._json                  MetaCoin._properties            MetaCoin._property_values
MetaCoin.abi                    MetaCoin.addProp                MetaCoin.address                MetaCoin.arguments
MetaCoin.ast                    MetaCoin.at                     MetaCoin.autoGas                MetaCoin.binary
MetaCoin.bytecode               MetaCoin.caller                 MetaCoin.class_defaults         MetaCoin.clone
MetaCoin.compiler               MetaCoin.configureNetwork       MetaCoin.contractName           MetaCoin.contract_name
MetaCoin.currentProvider        MetaCoin.decodeLogs             MetaCoin.defaults               MetaCoin.deployed
MetaCoin.deployedBinary         MetaCoin.deployedBytecode       MetaCoin.deployedSourceMap      MetaCoin.detectNetwork
MetaCoin.devdoc                 MetaCoin.ens                    MetaCoin.events                 MetaCoin.gasMultiplier
MetaCoin.hasNetwork             MetaCoin.interfaceAdapter       MetaCoin.isDeployed             MetaCoin.legacyAST
MetaCoin.length                 MetaCoin.link                   MetaCoin.links                  MetaCoin.metadata
MetaCoin.name                   MetaCoin.network                MetaCoin.networkType            MetaCoin.network_id
MetaCoin.networks               MetaCoin.new                    MetaCoin.numberFormat           MetaCoin.prototype
MetaCoin.resetAddress           MetaCoin.schemaVersion          MetaCoin.schema_version         MetaCoin.setNetwork
MetaCoin.setNetworkType         MetaCoin.setProvider            MetaCoin.setWallet              MetaCoin.source
MetaCoin.sourceMap              MetaCoin.sourcePath             MetaCoin.timeoutBlocks          MetaCoin.toJSON
MetaCoin.transactionHash        MetaCoin.unlinked_binary        MetaCoin.updatedAt              MetaCoin.updated_at
MetaCoin.userdoc                MetaCoin.web3
```

Figure 15.15: Available methods

3. Other methods can also be called in order to interact with the contract; for example, in order to retrieve the address of the contract, the following method can be called using the `truffle console`:

   ```
   truffle(development)> MetaCoin.address
   '0xeA0DB2Cee9093D7eA4230C63B55AA1cD79739c7C'
   ```

This address is also shown in contract creation transaction in Ganache:



```
CURRENT BLOCK  GAS PRICE    GAS LIMIT  HARDFORK    NETWORK ID  RPC SERVER             MINING STATUS  WORKSPACE
29             20000000000  6721975    PETERSBURG  5777        HTTP://127.0.0.1:7545  AUTOMINING     JAZZY-GIRL

TX 0x3fa67e71cbfb14de5cbe1e3eb72ddc7adfc4609e03feef23afc91b0b3b3428e0

SENDER ADDRESS                                  CREATED CONTRACT ADDRESS
0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA      0xeA0DB2Cee9093D7eA4230C63B55AA1cD79739c7C

VALUE        GAS USED     GAS PRICE      GAS LIMIT     MINED IN BLOCK
0.00 ETH     340697       20000000000    6721975       28

TX DATA
0x608060405234801561001057600080fd5b506127106000803273ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1
68152602001908152602001600020819055506103d6806100656000396000f3fe608060405234801561001057600080fd5b50600436106100415760003560e01c80637bd703
e814610046578063 90b98a111461009e578063f8b2cb4f14610104575b600080fd5b6100886004803603602081101561005c57600080fd5b81019080803573ffffffffffff
ffffffffffffffffffffffffffff16906020019092919050505061015c565b6040518082815260200191505060405180910390f35b6100ea6004803603604081101561 00b457
600080fd5b81019080803573ffffffffffffffffffffffffffffffffffffffff16906020019092919080359060200190929190505050610200565b60405180821515151515815
260200191505060405180910390f35b610146600480360360208110156101 1a57600080fd5b81019080803573ffffffffffffffffffffffffffffffffffffffff1690602001
909291905050506103595 65b6040518082815260200191505060405180910390f35b60007358e62cb54a8b0bd8bdc61061f24a98d88683197c6396e4ee3d610181846103595
65b600260405183 63ffffffff1660e01b815260040180838152602001828152602001925050506020604051808303818680 3b158015610 1be57600080fd5b505af415801561
01d2573d6000803e3d6000fd5b50505050604051 3d602081101561 01e857600080fd5b81019080805190602001909291905050509050919050565b6000816000803373fffff
ffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152602001600020541015610251576000905061035356
5b816000803373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002060008282540
392505081905550816000808573ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526020019081526020016000
20600082825401925050819055508273ffffffffffffffffffffffffffffffffffffffff163373ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b6
9c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef84604051808281526020019150506040518091039 0a3600190505b92915050565b60008060008373ffffffffffff
ffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054905091905056fea265627a7a72315820082
154cc060ed91bfe3d779a7ae6aa97d82143cdaad5f3cc85404ec004cc4d7464736f6c63430005100032

EVENTS

NO EVENTS
```

Figure 15.16: Contract creation transaction shown in Ganache

A few examples of other methods that we can call in the `truffle console` are shown here.

4. To query the accounts available, enter the following command:

```
truffle(development)> MetaCoin.web3.eth.getAccounts()
```

This will return the output shown here:

```
[ '0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA',
  '0x6805940005154aEdfe6a00A39C588ED668E64D9D',
  '0x56C21294F4e17dF32486b3d4D12E72D023861edF',
  '0xAfACDB553412071bB538E36d57ED92d6745C5f94',
  '0x694AE93a42C43B8E3d10c3F6769ADf2566C1863B',
  '0x52eA83ae62213f076E3A93054F8168907AaB3313',
  '0x4d4AAD022169bE203052173C09Da3027B66258b9',
  '0xC87B00b4105bc55FD38E6EC07d242Fa4906733b8',
  '0x605203fF0e161d9730F535Fa91FA6E095FaB8462',
  '0xCf5A1df3cc67eBAa085EcA037940671e81D4DB82' ]
```

5. To query the balance of the contract, enter the following command:

   ```
   truffle(development)> MetaCoin.web3.eth.getBalance("0x2366e9848803cB00CB8
   2E6E6De3F6D17C4AA9ADA")
   ```

   This will show the output shown here:

   ```
   '999457007800000000000'
   ```

   This is the first account shown in Ganache in the preceding screenshot, `0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA`, and the output returned a string with the value `'999457007800000000000'`.

6. To end a session with the `truffle console`, the `.exit` command is used.

This completes our introduction to the sample webpack Truffle box and the MetaCoin application using Truffle. In this section, we discovered how Truffle can be used to interact with the deployed smart contracts. MetaCoin in this section is an example of a decentralized application, however we used this merely as an example to learn how Truffle works. In this chapter's bonus content pages, we will use the techniques that we learned in this section to develop our own decentralized application.

In the next section, we will see how a contract can be developed from scratch, tested and deployed using Truffle, Ganache, and our privatenet.

Using Truffle to test and deploy smart contracts

This section will demonstrate how we can use Truffle for testing and deploying smart contracts. Let's look at an example of a simple contract in Solidity, which performs addition. We will see how migrations and tests can be created for this contract with the following steps.

1. Create a directory named `simple`:

   ```
   $ mkdir simple
   ```

2. Change directory to `simple`:

   ```
   $ cd simple
   ```

3. Initialize Truffle to create a skeleton structure for smart contract development:

   ```
   $ truffle init
   ✓ Preparing to download box
   ✓ Downloading
   ✓ cleaning up temporary files
   ✓ Setting up box
   ```

   The tree structure produced by the `init` command is as follows:

   ```
   .
   ├── contracts
   │   └── Migrations.sol
   ├── migrations
   │   └── 1_initial_migration.js
   ```

```
├── test
└── truffle-config.js

3 directories, 3 files
```

4. Place the two files `Addition.sol` and `Migrations.sol` in the `contracts` directory. The code for both of these files is listed as follows:

   `Addition.sol`:

```
pragma solidity ^0.5.0;

    contract Addition
    {
    uint8 x; //declare variable x

    // define function addx with two parameters y and z, and modifier public
    function addx(uint8 y, uint8 z ) public
    {
    x = y + z; //performs addition
    }
    // define function retrievex to retrieve the value stored, variable x
    function retrievex() view public returns (uint8)
    {
    return x;
    }
    }
```

   `Migrations.sol`:

```
pragma solidity >=0.4.21 <0.7.0;

contract Migrations {
  address public owner;
  uint public last_completed_migration;

  constructor() public {
    owner = msg.sender;
  }

  modifier restricted() {
    if (msg.sender == owner) _;
  }

  function setCompleted(uint completed) public restricted {
    last_completed_migration = completed;
  }
}
```

5. Now compile the contracts:

```
$truffle compile

Compiling your contracts...
===========================
> Compiling ./contracts/Additions.sol
> Compiling ./contracts/Migrations.sol
> Artifacts written to /Users/drequinox/simple/build/contracts
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang
```

6. Under the `migration` folder, place two files `1_initial_migration.js` and `2_deploy_contracts.js` as shown here:

   `1_initial_migration.js`:

```
var Migrations = artifacts.require("./Migrations.sol");

module.exports = function(deployer) {
deployer.deploy(Migrations);
};
```

   `2_deploy_contracts.js`:

```
var SimpleStorage = artifacts.require("Addition");

module.exports = function(deployer) {
deployer.deploy(SimpleStorage);
};
```

7. Under the `test` folder, place the file `TestAddition.sol`. This will be used for unit testing:

   `TestAddition.sol`:

```
pragma solidity ^0.4.2;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/Addition.sol";

contract TestAddition {

  function testAddition() public {
    Addition adder = Addition(DeployedAddresses.Addition());
    adder.addx(100,100);
    uint returnedResult = adder.retrievex();
    uint expected = 200;
    Assert.equal(returnedResult, expected, "should result 200");
  }
```

8. The test is run using the command shown here:

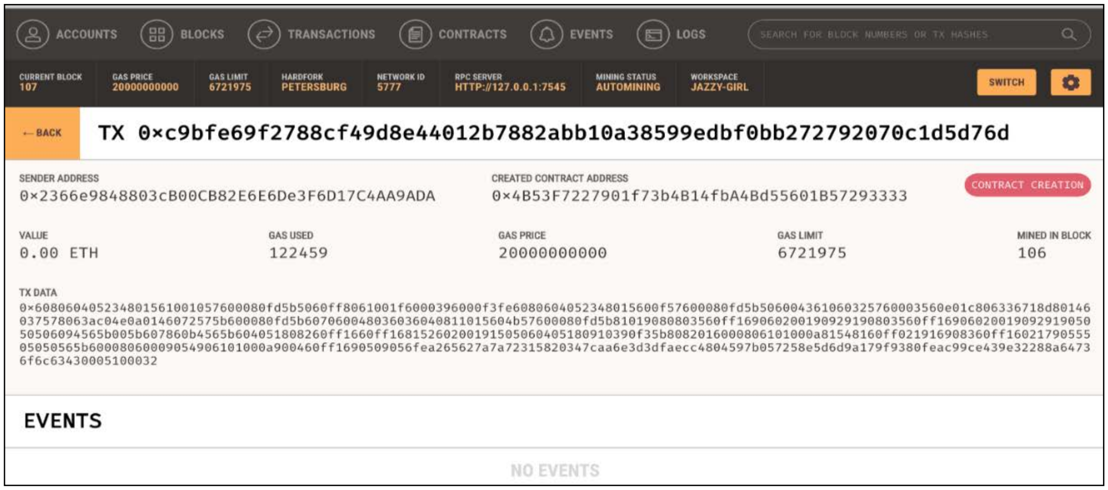
```
$truffle test
Using network 'development'.

Compiling your contracts...
===========================
> Compiling ./contracts/Addition.sol
> Compiling ./test/TestAddition.sol
> Artifacts written to /var/folders/82/5r_y_y_13wq4nqb0fw6s52nc0000gn
/T/test-202018-9102-1guyy3e.5s76
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

TruffleConfig {
  _deepCopy: [ 'compilers' ],
.
.  . . . . . . . . Text not shown for brevity . . . . . . . . . .
.

TestAddition
    ✓ testAddition (313ms)


  1 passing (9s)
```

9. This means that our tests are successful. Once the tests are successful, we can deploy it to the network, in our case, Ganache:

```
$ truffle migrate

Compiling your contracts...
===========================
> Compiling ./contracts/Addition.sol
> Artifacts written to /Users/drequinox/simple/build/contracts
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

Warning: Both truffle-config.js and truffle.js were found. Using truffle-
config.js.
Warning: Both truffle-config.js and truffle.js were found. Using truffle-
config.js.
```

```
Starting migrations...
======================
> Network name:    'development'
> Network id:      5777
> Block gas limit: 0x6691b7


2_deploy_contracts.js
=====================

   Deploying 'Addition'
   --------------------
   > transaction hash:
0xc9bfe69f2788cf49d8e44012b7882abb10a38599edbf0bb272792070c1d5d76d
   > Blocks: 0            Seconds: 0
   > contract address:    0x4B53F7227901f73b4B14fbA4Bd55601B57293333
   > block number:        106
   > block timestamp:     1581187718
   > account:             0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA
   > balance:             98.82759464
   > gas used:            122459
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00244918 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00244918 ETH


Summary
=======
> Total deployments:   1
> Final cost:          0.00244918 ETH
```

In Ganache we see this activity, which corresponds to our migration activity:



CURRENT BLOCK 107 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK PETERSBURG | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING | WORKSPACE JAZZY-GIRL

TX 0×c9bfe69f2788cf49d8e44012b7882abb10a38599edbf0bb272792070c1d5d76d

SENDER ADDRESS
0×2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA

CREATED CONTRACT ADDRESS
0×4B53F7227901f73b4B14fbA4Bd55601B57293333

VALUE
0.00 ETH

GAS USED
122459

GAS PRICE
20000000000

GAS LIMIT
6721975

MINED IN BLOCK
106

TX DATA

```
0×6080604052348015610010576000080fd5b5060ff8061001f6000396000f3fe6080604052348015600f57600080fd5b5060043610603257600003560e01c806336718d80146
037578063ac04e0a0146072575b600080fd5b6070600480360360408110156 04b57600080fd5b81019080803560ff169060200190929190803560ff16906020019092919050
50506094565b005b607860b4565b604051808260ff1660ff16815260200191505060405180910390f35b808201600080610100 0a81548160ff021916908360ff16021790555
05050565b6000806000905490610100 0a900460ff1690509056fea265627a7a72315820347caa6e3d3dfaecc4804597b057258e5d6d9a179f9380feac99ce439e32288a6473
6f6c63430005100032
```

EVENTS

NO EVENTS

Figure 15.17: Ganache deployed contract—contract creation transaction

As the `Addition` contract is already instantiated and available in the `truffle console`, it becomes quite easy to interact with the contract. In order to interact with the contract, the following methods can be used:

1. Run the following command:

   ```
   $ truffle console
   ```

   This will open the `truffle console`, which will allow interaction with the contract. For example, in order to retrieve the address of the deployed contract, the following method can be called:

   ```
   truffle(development)> Addition.address
   '0x4B53F7227901f73b4B14fbA4Bd55601B57293333'
   ```

2. To call the functions from within the contract, the deployed method is used with the contract functions. An example is shown here. First instantiate the contract:

   ```
   truffle(development)> let additioncontract = await Addition.deployed()
   undefined
   ```

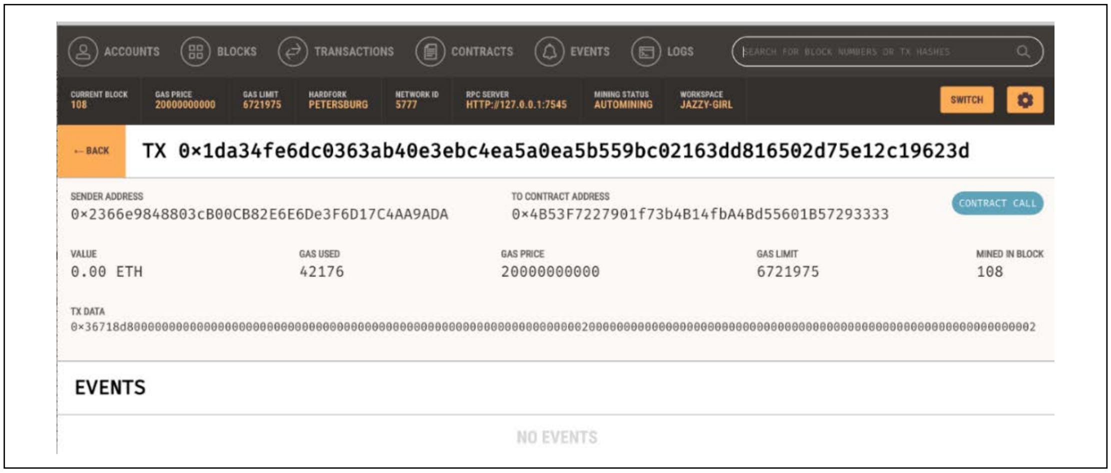
3. Now call the `addx` function:

   ```
   truffle(development)> additioncontract.addx(2,2)
   ```

This will produce the following output indicating the execution status of the transaction that was created as a result of calling the `addx` function:

```
{ tx:

'0x1da34fe6dc0363ab40e3ebc4ea5a0ea5b559bc02163dd816502d75e12c19623d',
  receipt:
   { transactionHash:

'0x1da34fe6dc0363ab40e3ebc4ea5a0ea5b559bc02163dd816502d75e12c19623d',
     transactionIndex: 0,
     blockHash:

'0xe57b0427f364f774094479f1f9ebd2b46f8f5d00a341b46ea164644eb1771bb5',
     blockNumber: 108,
     from: '0x2366e9848803cb00cb82e6e6de3f6d17c4aa9ada',
     to: '0x4b53f7227901f73b4b14fba4bd55601b57293333',
     gasUsed: 42176,
     cumulativeGasUsed: 42176,
     contractAddress: null,
     logs: [],
     status: true,
     logsBloom:
      '0x00000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
0000000000000000000',
     v: '0x1c',
     r:

'0xc74a0de80218f699ad9a84f7017feaa9f326337373203d10fabde7e4c920dc6f',
     s:

'0x4749fd0fc2631dcb64c712cf2a3818c6375b4fe863860b49d07b673cbc729afb',
     rawLogs: [] },
  logs: [] }
```

We see the transaction in Ganache too:



CURRENT BLOCK 108 GAS PRICE 20000000000 GAS LIMIT 6721975 HARDFORK PETERSBURG NETWORK ID 5777 RPC SERVER HTTP://127.0.0.1:7545 MINING STATUS AUTOMINING WORKSPACE JAZZY-GIRL

TX 0x1da34fe6dc0363ab40e3ebc4ea5a0ea5b559bc02163dd816502d75e12c19623d

SENDER ADDRESS
0x2366e9848803cB00CB82E6E6De3F6D17C4AA9ADA

TO CONTRACT ADDRESS
0x4B53F7227901f73b4B14fbA4Bd55601B57293333

VALUE 0.00 ETH GAS USED 42176 GAS PRICE 20000000000 GAS LIMIT 6721975 MINED IN BLOCK 108

TX DATA
0x36718d8000000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000002

EVENTS

NO EVENTS

Figure 15.18: Transaction in Ganache

4. Finally, we can call the `retrievex` function to see the current value after `addition`:

```
truffle(development)> additioncontract.retrievex()
<BN: 4>
```

Finally, we see the value `4` returned by the contract.

In this section, we created a simple smart contract that performs an addition function and learned how the Truffle framework can be used to test and deploy the smart contract. We also learned how to interact with the smart contract using the Truffle console.

In the next section, we will look at IPFS, which can serve as the decentralized storage layer for a decentralized ecosystem. We will now see how we can host one of our DApps on IPFS.

Deployment on decentralized storage using IPFS

As discussed in *Chapter 1*, *Blockchain 101*, in order to fully benefit from decentralized platforms, it is desirable that you decentralize the storage and communication layer too in addition to decentralized state/computation (blockchain). Traditionally, web content is served via centralized servers, but that part can also be decentralized using distributed filesystems. The HTML content shown in the earlier examples can be stored on a distributed and decentralized IPFS network in order to achieve enhanced decentralization.

IPFS is available at `https://ipfs.io/`.

Note that IPFS is under heavy development and is an alpha release. Therefore, there is the possibility of security bugs. Security notes are available here: `https://ipfs.io/ipfs/QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG/security-notes`

IPFS can be installed by following this process:

1. Download the IPFS package from `https://dist.ipfs.io/#go-ipfs` and decompress the `.gz` file:

   ```
   $ tar xvfz go-ipfs_v0.4.23_darwin-amd64.tar
   ```

2. Change directory to where the files have been decompressed:

   ```
   $ cd go-ipfs
   ```

3. Start the installation:

   ```
   $ ./install.sh
   ```

   This will produce the following output:

   ```
   Moved ./ipfs to /usr/local/bin
   ```

4. Check the version to verify the installation:

   ```
   $ ipfs version
   ```

   This will produce the following output:

   ```
   ipfs version 0.4.23
   ```

5. Initialize the IPFS node:

   ```
   $ ipfs init
   ```

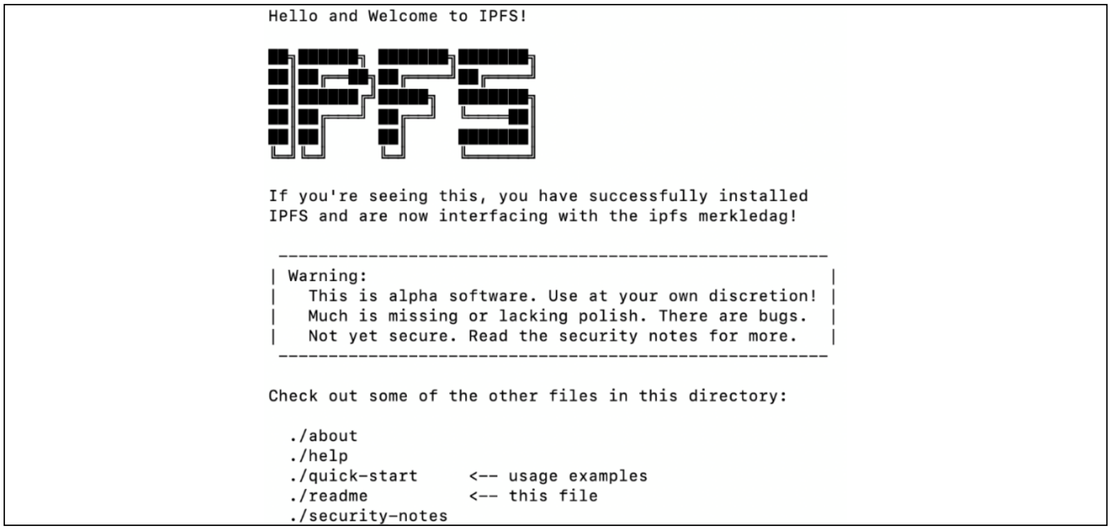
   This will produce the following output:

   ```
   initializing IPFS node at /Users/drequinox/.ipfs
   generating 2048-bit RSA keypair...done
   peer identity: QmSxkXkCwqM2qbFoxMEfjbk9w17zofXFin4ZeuxDcPRe5g
   to get started, enter:
   ```

6. Enter the following command to ensure that IPFS has been successfully installed:

   ```
   $ ipfs cat /ipfs/QmS4ustL54uo8FzR9455qaxZwuMiUhyvMcX9Ba8nUH4uVv/readme
   ```

This will produce the following output:



```
Hello and Welcome to IPFS!

If you're seeing this, you have successfully installed
IPFS and are now interfacing with the ipfs merkledag!

 -------------------------------------------------------
| Warning:                                              |
|   This is alpha software. Use at your own discretion! |
|   Much is missing or lacking polish. There are bugs.  |
|   Not yet secure. Read the security notes for more.   |
 -------------------------------------------------------

Check out some of the other files in this directory:

  ./about
  ./help
  ./quick-start     <-- usage examples
  ./readme          <-- this file
  ./security-notes
```

Figure 15.19: IPFS installation

7. Now, start the IPFS daemon:

```
$ ipfs daemon
```

This will produce the following output:

```
Initializing daemon...
go-ipfs version: 0.4.23-
Repo version: 7
System version: amd64/darwin
Golang version: go1.13.7
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/192.168.0.18/tcp/4001
Swarm listening on /ip6/::1/tcp/4001
Swarm listening on /p2p-circuit
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip4/192.168.0.18/tcp/4001
Swarm announcing /ip4/82.2.27.41/tcp/4001
Swarm announcing /ip6/::1/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8080
Daemon is ready
```

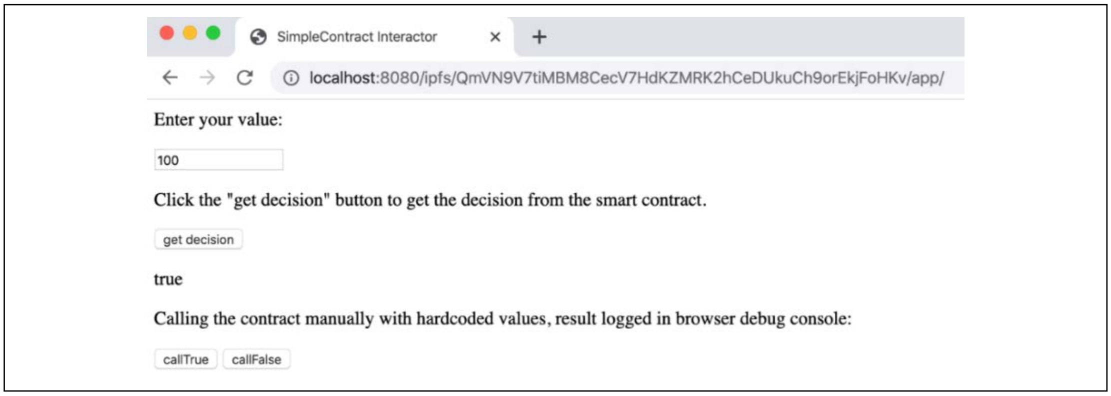
8. Copy files onto IPFS using the following command:

```
$ ipfs add . --recursive -progress
```

This will produce the following output, indicating the progress of adding the pages to IPFS:

```
added QmczxMDDHXvxr97XZMsaf3zZYUZL9fUdiTVQUjYJnpAy7C
simplecontract/app/app.js
added QmSjKPg7L52x33JiKsjDaVDXbK2ntxY75w7w6TBMd9pN6i
simplecontract/app/index.html
added QmS9SkKyt6ZdjBxnK36rbgWh46Q17A2MK3cuNYzKeKhcZc
simplecontract/app/web3.js
added QmfSKCZ7NR5zBdfbjdUfFFuAEvZLiqPGUZwYog8u1Zg1XJ
simplecontract/build/contracts/Addition.json
added QmdVnjZvZXiJS7H7jbsKUmu4JVyz6ftSPpggy7q13w2Q8q
simplecontract/build/contracts/Migrations.json
added QmWrnrBaV5ksG5E2yyvM6iZV73iwfG7tvv6MrN716286Bh
simplecontract/contracts/Addition.sol
added QmZdUWWgL2DD6JMBszyju5ECkbDN66UTQMQPLCQ3kGATZj
simplecontract/contracts/Migrations.sol
added QmcxaWKQ2yPkQBmnpXmcuxPZQUMuUudmPmX3We8Z9vtAf7
simplecontract/migrations/1_initial_migration.js
added QmUfCSF2HxrscKRAd7Z6NrRTo9QnkFfDqLavYQDtGD7Wqs
simplecontract/migrations/2_deploy_contracts.js
added QmSCf4G68ZC6EWJpyE4E2kLn1LuSs3EbdZLJ27eDkLZPH3
simplecontract/test/TestAddition.sol
added QmfRLQWpu9tjrgX4p9nJHEWLxzdMqBQZDnxVnwYPPPEjNK
simplecontract/truffle-config.js
added QmXatyNNTYubxjRpMiZHEUz6yiqgDmBDaVXjnM37h3HUAQ
simplecontract/truffle.js
added QmXT5f7TPv1BU4XbVgXYutdMeuEBvo9C7fzshmyTpnx1D5
simplecontract/app
added QmPbEVcrjsd8xkE4ePJ6hNN1C9yPshLR8moAft3w3MtMRU
simplecontract/build/contracts
added QmTDHJX5UwMCYwoHj3hQ32E7xtKmpXg9ZhDXHntGzshdwB
simplecontract/build
added QmQGyeiRY9MwX3E8JjubAoziaqQf9QNH4RXeZZzN5xLH12
simplecontract/contracts
added QmQmSgckTHGN6oCZ7YFrd5mHQGSucNt1gLjDoAZca54n5p
simplecontract/migrations
added QmUgAfZs6DfdgqfcCLYGAFdPZ5RRoLw2ma2Y8fvvP8y4do
simplecontract/test
added QmVN9V7tiMBM8CecV7HdKZMRK2hCeDUkuCh9orEkjFoHKv
simplecontract
```

Now it can be accessed in the browser as follows:


SimpleContract Interactor
localhost:8080/ipfs/QmVN9V7tiMBM8CecV7HdKZMRK2hCeDUkuCh9orEkjFoHKv/app/

Enter your value:

100

Click the "get decision" button to get the decision from the smart contract.

get decision

true

Calling the contract manually with hardcoded values, result logged in browser debug console:

callTrue callFalse

Figure 15.20: Example Truffle DApp running on IPFS and served via web host

Note that the URL is pointing to the IPFS filesystem, `http://localhost:8080/ipfs/QmVN9V7tiMBM8CecV7HdKZMRK2hCeDUkuCh9orEkjFoHKv/app/`.

9. Finally, in order to make the changes permanent, the following command can be used:

```
$ ipfs pin add QmVN9V7tiMBM8CecV7HdKZMRK2hCeDUkuCh9orEkjFoHKv
```

This will show the following output:

```
pinned QmVN9V7tiMBM8CecV7HdKZMRK2hCeDUkuCh9orEkjFoHKv recursively
```

The preceding example demonstrated how IPFS can be used to provide decentralized storage for the web part (UI) of smart contracts.

Remember that in *Chapter 1*, *Blockchain 101*, we described that a decentralized application consists of a frontend interface (usually a web interface), backend smart contracts on a blockchain, and the underlying blockchain. We have covered all these elements in this example and created a decentralized application.

To try your hand at a start-to-finish application deployment project, please go to this book's bonus online content pages here: `https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf`. You will build and deploy a proof of idea contract using Truffle, before creating a UI frontend for it with a tool called Drizzle!

Summary

This chapter started with the introduction of Web3. We explored various methods to develop smart contracts. Also, we saw how the contract can be tested and verified using local test blockchain before implementation on a public blockchain or private production blockchain.

We worked with various tools such as Ganache, the Geth client console, and the Remix IDE to develop, test, and deploy smart contracts. Moreover, the Truffle framework was also used to test and migrate smart contracts. We also explored how IPFS can be used to host the webpages that we created for our DApp, serving as the decentralized storage layer of the blockchain ecosystem.

In the bonus content for this chapter, which we strongly encourage you to use, we practiced the techniques we learned in this chapter, plus other advanced topics such as Drizzle, to create the frontends for DApps easily.

In the next chapter, we will introduce Serenity, Ethereum 2.0, which is the final version of Ethereum.