CHAPTER 5

Blockchain Application Development

In the previous chapters we went into theoretical details about what blockchain is and how the Bitcoin and Ethereum blockchains work. We also looked at the different cryptographic and mathematical algorithms, theorems, and proofs that go into making the blockchain technology.

In this chapter, we will start with how blockchain applications are different than the conventional applications, and then we will dive into how to build applications on blockchains. We will also look at setting up the necessary infrastructure needed to start developing decentralized applications.

Decentralized Applications

The popularity of blockchain technology is mostly driven by the fact that it can potentially solve various real-world problems because it provides more transparency and security (tamper-proof) than conventional technologies. There are a lot of blockchain use cases identified by several startups and community members aimed at solving these problems.

To implement these use cases, we create applications that work on top of blockchains. In general, applications that interact with blockchains are referred to as "decentralized applications" or, in short, just DApps or dApps.

To understand DApps better, let's first revisit what a blockchain is. A blockchain or a distributed ledger is basically a special kind of database where the data is not stored at a centralized server, but it is copied at all the participating nodes in the network. Also, the data on blockchains is cryptographically signed, which proves the identity of the entity who wrote that data on the blockchain. To make use of this database to store and retrieve data, we create applications that are called DApps because these applications do not rely on a centralized database but on a blockchain-based decentralized data store. There is no single point of failure or control for these applications.

Let's take an example of a DApp. Let's take a scenario of supply chain where several vendors and logistics partners are involved in the supply chain process of manufactured goods. To use blockchain technology for this supply chain use case, here's what we would do:

- We would need to set up blockchain nodes at each of these vendors so that they can participate in the consensus process on the data shared.
- We would need an interface so that all the participants and users can store, retrieve, verify, and evaluate data on the blockchain. This interface would be used by the manufacturer to enter the information about the goods manufactured; by the logistics partner to enter information about the transfer of goods; by the warehousing vendor to verify if the goods manufactured and the goods transferred are in sync, etc., etc. This interface would be our supply chain DApp.

Another example of a DApp would be a voting system based on blockchains. Using blockchain for voting, we would be able to make the whole process much more transparent and secure because each vote would be cryptographically signed. We would need to create an application that could get a list of candidates for whom voters could vote, and this application would also provide a simple interface to submit and record the votes.

Blockchain Application Development

Before we jump into code, let's first understand some basic concepts around blockchain application development. Generally, we are used to concepts like objects, classes, functions, etc. when we develop conventional software applications. However, when it comes to blockchain applications, we need to understand a few more concepts like *transactions, accounts and addresses, tokens and wallets, inputs,* and *outputs and balances*. The handshake and request/response mechanism between a decentralized application and a blockchain are driven by these concepts.

First, when developing an application based on blockchain, we need to identify how the application data would map to the blockchain data model. For example, when developing a DApp on the Ethereum blockchain, we need to understand how the application state can be represented in terms of Solidity data structures and how the application's behavior can be expressed in terms of Ethereum smart contracts. As we know that all data on a blockchain is cryptographically signed by private keys of the users, we need to identify which entities in our application would have identities or addresses represented on the blockchain. In conventional applications this is generally not the case, because the data is not always signed. For blockchain application we need to define who would be the signers and what data they would sign. For example, in a voting DApp in which every voter cryptographically signs their vote, this is easy to identify. However, imagine a scenario where we need to migrate an existing conventional distributed systems application, having its data stored across multiple SQL tables and databases, to a DApp based on Ethereum blockchain. In this case we need to identify which entities in which table would have their identities and which entities would be *attached* to other identities.

In the next few sections, we will explore Bitcoin and Ethereum application programming using simple code snippets to send some transactions. The purpose of this exercise is to become familiar with the blockchain APIs and common programming practices. For simplicity, we will be using public test networks for these blockchains and we will write code in JavaScript. The reason for selecting JavaScript is, at the time of this writing, we have stable JavaScript libraries available for both blockchains and it will be easier to understand the similarities and differences in the approaches we take while writing code. The code snippets are explained in detail after every logical step and can be understood even if the reader is not familiar with JavaScript programming.

Libraries and Tools

Recall from Chapter 2, that there are a lot of cryptographic algorithms and mathematics used in blockchain technology. Before we send our transactions to blockchains from an application, we need to prepare them. The transaction preparation includes defining accounts and addresses, adding required parameters and values to the transaction objects, and signing using private keys, among a few other things. When developing applications, it's better to use verified and tested libraries for transaction preparation instead of writing code from scratch. Some of the stable libraries for both Bitcoin and Ethereum are available open source, which can be used to prepare and sign transactions and to send them to the blockchain nodes/network. For the purpose of our code exercises, we will be using the *bitcoinjs* JavaScript library for interacting with the

Bitcoin blockchain and the *web3.js* JavaScript library for interacting with the Ethereum blockchain. Both these libraries are available as node.js packages and can be downloaded and integrated using the *npm* commands.

Important Note The code exercises in this chapter are based on **node.js** applications. This is to make sure that the code we write as part of this exercise has a container in which it can run and interact with the other prepackaged libraries (node modules) mentioned. It is nice to have some knowledge about node.js application development, and the reader is encouraged to follow a *getting started* tutorial on *node.js* and *npm*.

Figure 5-1 shows how a DApp interacts with a blockchain.

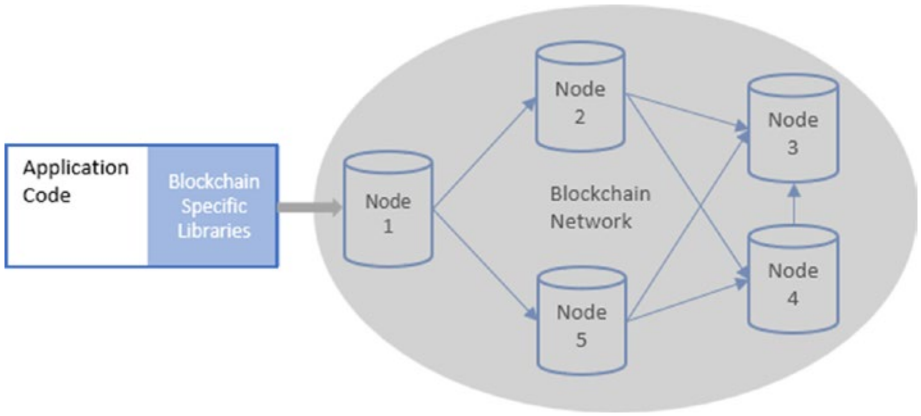


Figure 5-1. *Blockchain application interaction*

Interacting with the Bitcoin Blockchain

In this section we will send a transaction to the Bitcoin public test network from one address to another. Consider this a "*Hello World*" application for the Bitcoin blockchain. As mentioned before, we will be using the *bitcoinjs* JavaScript library for preparing and signing transactions. And for simplicity, instead of hosting a local Bitcoin node, we will use a public Bitcoin test network node hosted by a third-party provider *block-explorer*. Note that you can use any provider for your application and you can also host a local node. All you need to do is to point your application code to connect to your preferred node.

Recall from previous chapters that the Bitcoin blockchain is primarily for enabling peer to peer payments. A Bitcoin transaction is mostly just a transfer of Bitcoins from one address to another. Here's how we do this programmatically.

The following (Figure 5-2) shows how this code interacts with the Bitcoin blockchain. **Note:** The figure is just a rough sketch and does not show the Block Explorer service architecture in detail.

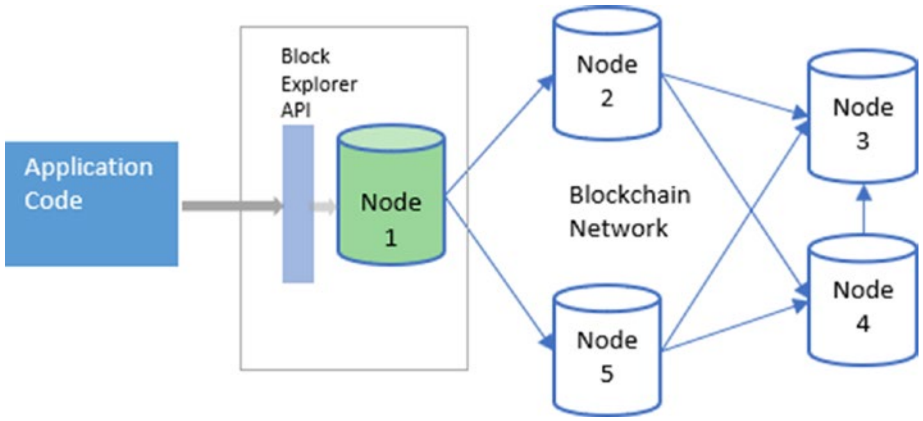


Figure 5-2. *Application interacting with the Bitcoin blockchain using the Block Explorer API*

The following subheadings of this section are steps to follow, in that order, to send a transaction to the Bitcoin test network using JavaScript.

Setup and Initialize the bitcoinjs Library in a *node.js* Application

Before we call the library-specific code for Bitcoin transactions, we will install and initialize the *bitcoinjs* library.

After initializing a node.js applicaion using the **`npm init`** command, let's create an entry point for our application, *index.js*, and custom JavaScript module to call the bitcoinjs library functions *btc.js*. Import *btc.js* in the *index.js*. Now, we are ready to follow the next steps.

First, let's install the node module for *bitcoinjs*:

```
npm install --save bitcoinjs-lib
```

Then, in our Bitcoin module btc.js, we will initialize the bitcoinjs library using the require keyword:

```
var btc = require('bitcoinjs-lib');
```

Now we can use this btc variable to call library functions on the bitcoinjs library. Also, as part of the initialization process, we are initializing a couple of more variables:

- The network to target : We are using the Bitcoin test network.

  ```
  var network = btc.networks.testnet;
  ```

- The public node API endpoint to get and post transactions : We are using the Block Explorer API for Bitcoin test network. Note that you can replace this API endpoint with your preferred one.

  ```
  var blockExplorerTestnetApiEndpoint =
  'https://testnet.blockexplorer.com/api/';
  ```

At this point, we are all set up to create a Bitcoin transaction using a node.js application.

Create Keypairs for the Sender and Receiver

The first thing that we will need are the keypairs for the sender and the receivers. These are like user accounts identifying the users on the blockchain. So, let's first create two keypairs for Alice and Bob.

```
var getKeys = function () {
    var aliceKeys = btc.ECPair.makeRandom({
        network: network
    });
    var bobKeys = btc.ECPair.makeRandom({
        network: network
    });
    var alicePublic = aliceKeys.getAddress();
    var alicePrivate = aliceKeys.toWIF();
    var bobPublic = bobKeys.getAddress();
    var bobPrivate = bobKeys.toWIF();
    console.log(alicePublic, alicePrivate, bobPublic,
    bobPrivate);
};
```

What we did in the previous code snippet is, we used the **ECPair** class of the bitcoinjs library and called the **makeRandom** method on it to create random keypairs for the test network; note the parameter passed for network type.

Now that we have created a couple of keypairs, let's use them to send Bitcoins from one to the other. In almost all the cryptography examples, Alice and Bob have been the favorite characters, as seen in the preceding keypair variables. However, every time we see a cryptography example, generally Alice is the one who encrypts/signs something and sends to Bob.

For that reason, we feel Bob is under a lot of debt from Alice, so in our case we will help Bob repay some of that debt. We will do this example Bitcoin transaction from Bob to Alice.

Get Test Bitcoins in the Sender's Wallet

We have identified that Bob is going to be acting as the sender in this example Bitcoin transaction. Before he sends any Bitcoins to Alice, he needs to own them. As we know that this example transaction is targeting the Bitcoin test network, there is no real money involved but we still need some test Bitcoins in Bob's wallet. A simple way to get test network Bitcoins is to ask on the Internet. There are a lot of websites on the Internet that host a simple web form to take the Bitcoin testnet addresses and then send test net Bitcoins to those. These services are called Bitcoin testnet faucets, and if you search online for that term you will get a lot of those in the search results. We are not listing or recommending any specific testnet faucet because they are generally not permanent. As soon as a faucet service provider has exhausted their test coins, or they don't want to host the service anymore, they shut it down. But then new ones keep coming up all the time. A list of some of these faucet services is also available on the Bitcoin wiki testnet page.

Another way of getting test net Bitcoins is to host a local Bitcoin node pointing to the test net and mine some. The block mining on the Bitcoin test network is not as difficult as that on the main network. This approach could well be the next level approach when you are building a production Bitcoin application and you need to test it frequently. Instead of asking for test coins every time you want to test your application, you can just mine them yourself.

For the purposes of this simple example, we will just get some Bitcoins from a testnet faucet. In the previous code snippet, the value in the bobPublic variable is Bob's Bitcoin testnet address. When we ran this snippet, it generated "**`msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2`**" as Bob's address. It is also Bob's base 58 encoded public key. We will submit this value in one of the testnet faucet web forms and in return we will receive a transaction ID. If we look up that transaction ID on any of the Bitcoin testnet explorers, we will see that some other address has sent some test Bitcoins to Bob's address we submitted in the form.

Get the Sender's Unspent Outputs

Now that we know that we have some test Bitcoins in Bob's wallet, we can spend them and give them to Alice through a Bitcoin transaction. Let's recall from Chapter 3 how the Bitcoin transactions are made of inputs and outputs. You can spend your unspent outputs by adding them as inputs to the transactions where you want to spend them. To do that, first you need to query the network about the sender's unspent outputs. Here's how we will do that for Bob's Bitcoin testnet address using the block explorer API. To get the unspent outputs, we will send an HTTP request to the UTXO endpoint with Bob's address `"msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2"`.

```
var getOutputs = function () {
    var url = blockExplorerTestnetApiEndpoint + 'addr/' +
    msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2 + '/utxo';
    return new Promise(function (resolve, reject) {
        request.get(url, function (err, res, body) {
            if (err) {
                reject(err);
            }
            resolve(body);
        });
    });
};
```

In the previous code snippet, we have used the node.js request module to send http requests using a node.js application. Feel free to use your favorite http library/module. This snippet is a JavaScript function that returns a promise that resolves into the response body from the API method. Here's how the response looks:

```
[
    {
        address: 'msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2',
        txid: 'db2e5966c5139c6e937203d567403867643482bbd9a6624
        752bbc583ca259958',
        vout: 0,
        scriptPubKey: '76a914806094191cbd4fcd8b4169a70588ad
        c51dc02d6888ac',
        amount: 0.99992,
        satoshis: 99992000,
        height: 1258815,
        confirmations: 1011
    },
    {
      address: 'msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2',
        txid: '5b88d5fc4675bb86b0a3a7fc5a36df9c425c3880a7
        453e3afeb4934e6d1d928e',
        vout: 1,
        scriptPubKey: '76a914806094191cbd4fcd8b4169a70588ad
        c51dc02d6888ac',
        amount: 0.99998,
        satoshis: 99998000,
        height: 1258814,
        confirmations: 1012
    }
]
```

The response body returned by the call is a JSON array with two objects. Each of these objects represents an unspent output for Bob. Each output has txid, which is the transaction ID where this output is listed, the amount associated with output, and the vout, which means the sequence or index number of the output in that transaction. There is some other information in the JSON objects too, but that will not be used in the transaction preparation process.

If we take the first object in the array, it basically says that the Bitcoin testnet address **"msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2"** has `` `99992000` `` unspent *satoshis* coming from the transaction `` `db2e5966c5139c6e937203d567403867643482bbd9a6624752bbc583ca259958` `` at the index `` `0` ``. Similarly, the second object represents `` `99998000` `` unspent satoshis coming from the transaction `` `5b88d5fc4675bb86b0a3a7fc5a36df9c425c3880a7453e3afeb4934e6d1d928e` `` at the index `` `1` ``.

Don't forget that **"msDkUzzd69idLLGCkDFDjVRz44jHcV3pW2"** is Bob's Bitcoin testnet, which we created in step 2 earlier. Now we know that Bob has this many satoshis, which he can spend in a new transaction.

Prepare Bitcoin Transaction

The next step is to prepare a Bitcoin transaction in which Bob can send the test coins to Alice. Preparing the transaction is basically defining its inputs, outputs, and amount.

As we know from the previous step that Bob has two unspent outputs under his Bitcoin testnet address, let's spend the first element of the outputs array. Let's add this as an input to our transaction.

```
var utxo = JSON.parse(body.toString());
var transaction = new btc.TransactionBuilder(network);
transaction.addInput(utxo[0].txid, utxo[0].vout);
```

In the prceding code snippet, first we have parsed the response we received from the previous API call to get Bob's unspent outputs.

Then we have created a transaction builder object for the Bitcoin test network using the bitcoinjs library.

In the last line, we have defined a transaction input. Note that this input is referring to the element at 0 index of the utxo array, which we received in the API call from the previous step. We have passed the transaction ID (txid) and vout from the unspent to the **`transaction.addInput`** method as input parameters.

Basically, we are defining what we want to spend and where we got it from.

Next, we add the transaction outputs. This is where we say how we want to spend what we added in the input. In the line following, we have added a transaction output by calling the **`addOutput`** method on the transaction builder object and passed in the target address and the amount. Bob wants to send 99990000 satoshis to Alice. Notice that we have used Alice's Bitcoin testnet address as the function's first parameter.

```
transaction.addOutput(alicePublic, 99990000);
```

While we have used only one input and one output in this example transaction, a transaction can have multiple inputs and outputs. An important thing to note is that the total amount in inputs should not be less than the total amount in outputs. Most of the time, the amount in inputs is slightly more than the amount in outputs, and the difference is the transaction fee offered to the miners to include this transaction when they mine the next block.

In this transaction, we have 2,000 satoshis as the transaction fee, which is the difference between input amount (99992000) and the output amount (99990000). Note that we don't have to create any outputs for the transaction fee; the difference between the input and output total amounts is automatically taken as the transaction fee.

Also, note that we cannot spend partial unspent outputs. If an unspent output has *x* amount of Bitcoins associated with it then we must spend all of the *x* Bitcoins when adding this unspent output as an input in a transaction. So, in case Bob doesn't want to give all the 99,990,000 satoshis associated with his unspent output to Alice, then we need to give it back to Bob by adding another output to the transaction with an amount equal to the difference of total unspent amount and the amount Bob wants to give to Alice.

Sign Transaction Inputs

Now, that we have defined the inputs and outputs in the transaction, we need to sign the inputs using Bob's keys. The following line of code calls the **sign** function on the transaction builder object to cryptographically sign the transaction using Bob's private key, but it takes the whole key pair object as an input parameter.

```
transaction.sign(0, bobKeys);
```

Note that the **transaction.sign** function takes the index of the input and the full key pair as input parameters. In this transaction, because we have only one input, the index we have passed is 0.

At this stage, our transaction is prepared and signed.

Create Transaction Hex

Now we will create a hex string from the transaction object.

```
var transactionHex = transaction.build().toHex();
```

The output of this line of code is the following string, which represents our prepared transaction; this step is needed because the send transaction API accepts the raw transaction as a string.

Broadcast Transaction to the Network

Finally, we use the hex string value we generated in the last step and send it to the block explorer public testnet node using the API,

```
var txPushUrl = blockExplorerTestnetApiEndpoint + 'tx/send';
request.post({
    url: txPushUrl,
        json: {
            rawtx: transactionHex
        }
    }, function (err, res, body) {
        if (err) console.log(err);

        console.log(res);
        console.log(body);
    });
```

If the transaction is accepted by the block explorer public node, we will receive a transaction ID as the response of this API call,

```
{
    txid: "db2e5966c5139c6e937203d567403867643482bbd
    9a6624752bbc583ca259958"
}
```

Now that we have the transaction ID of our transaction, we can look it up on any of the online testnet explorers to see if and when it gets mined and how many confirmations it has.

Putting it all together, here's the complete code for sending a Bitcoin testnet transaction using JavaScript. The input parameters are the Bitcoin testnet keypairs we created in step 1.

```
var createTransaction = function (aliceKeys, bobKeys) {
    getOutputs(bobKeys.getAddress()).then(function (res) {
        var utxo = JSON.parse(res.toString());
        var transaction = new btc.TransactionBuilder(network);
        transaction.addInput(utxo[0].txid, utxo[0].vout);
        transaction.addOutput(alicekeys.getAddress(),
        99990000);
        transaction.sign(0, bobKeys);
        var transactionHex = transaction.build().toHex();
        var txPushUrl = blockExplorerTestnetApiEndpoint +
        'tx/send';
        request.post({
            url: txPushUrl,
            json: {
                rawtx: transactionHex
            }
        }, function (err, res, body) {
            if (err) console.log(err);

            console.log(res);
            console.log(body);
        });
    });
};
```

In this section we learned how we can programmatically send a transaction to the Bitcoin test network. Similarly, we can send transactions to the Bitcoin main network by using the main network as the target in the library functions and in the API endpoints. We also used the query APIs to get unspent outputs of a Bitcoin address. These functions can be used to create a simple Bitcoin wallet application to query and manage Bitcoin addresses and transactions.

Interacting Programmatically with Ethereum—Sending Transactions

The Ethereum blockchain has much more to offer in terms of blockchain application development as compared with the Bitcoin blockchain. The ability to execute logic on the blockchain using smart contracts is the key feature of Ethereum blockchain that allows developers to create decentralized applications. In this section we will learn how to programmatically interact with the Ethereum blockchain using JavaScript. We will look at the main aspects of Ethereum application programming from simple transactions to creating and calling smart contracts.

As we did for interacting with the Bitcoin blockchain in the previous section, we will be using a JavaScript library and test network for interacting with Ethereum as well. We will use the web3 JavaScript library for Ethereum. This library wraps a lot of Ethereum JSON RPC APIs and provides easy to use functions to create Ethereum DApps using JavaScript. At the time of this writing, we are using a version greater than and compatible with version 1.0.0-beta.28 of the web3 JavaScript library.

For the test network, we will be using the *Ropsten* test network for Ethereum blockchain.

For simplicity, we will again use a public-hosted test network node for Ethereum so that we don't have to host a local node while running these code snippets. However, all snippets should work with a locally hosted node as well. We are using the Ethereum APIs provided by the Infura service. Infura is a service that provides public-hosted Ethereum nodes so that developers can easily test their Ethereum apps. There is a small and free registration step needed before we can use the Infura API, so we will go to `https://infura.io` and do a registration. We will get an API key after registration. Using this API key, we can now call the Infura API.

The following (Figure 5-3) shows how this code interacts with the Ethereum blockchain. **Note:** The figure is just a rough sketch and does not show the Infura service architecture in detail.

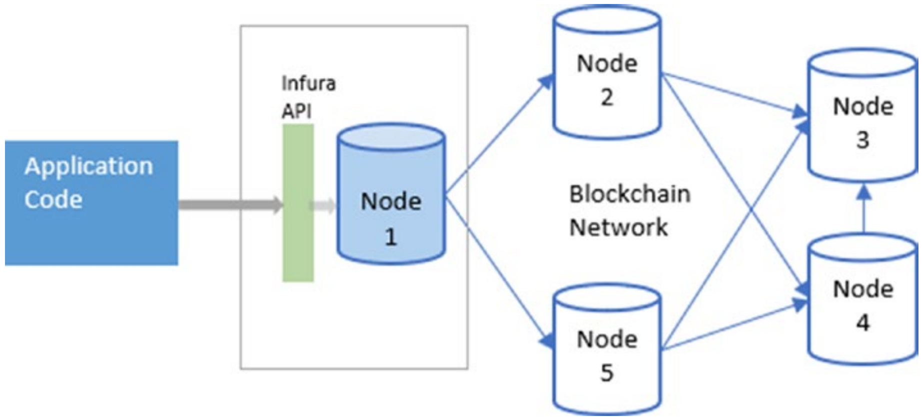


Figure 5-3. *Application interacting with Ethereum blockchain using Infura API service*

The following subsections of this section are steps to follow, in that order, to send a transaction to the Ethereum Ropsten test network using JavaScript.

Set Up Library and Connection

First, we install the web3 library in our node.js application. Note the specific version of library mentioned in the installation command. This is because version 1.0.0 of the library has some more APIs and functions available and they reduce dependency on other external packages.

```
npm install web3@1.0.0-beta.28
```

Then, we initialize the library in our nodejs Ethereum module using the require keyword,

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

Now, we have a reference of the web3 library, but we need to instantiate it before we can use it. The following line of code creates a new instance of the Web3 object and it sets the Infura-hosted Ethereum Ropsten test network node as the provider for this Web3 instance.

```
var web3 = new Web3(new Web3.providers.HttpProvider('https://
ropsten.infura.io/<your Infura API key>'));
```

Set Up Ethereum Accounts

Now that we are all set up, let's send a transaction to the Ethereum blockchain. In this transaction, we will send some Ether from one account to another. Recall from Chapter 4 that Ethereum does not use the UTXO model but it uses an account and balances model.

Basically, the Ethereum blockchain manages state and assets in terms of accounts and balances just like banks do. There are no inputs and outputs here. You can simply send Ether from one account to another and Ethereum will make sure that the states are updated for these accounts on all nodes.

To send a transaction to Ethereum that transfers Ether from one account to others, we will first need a couple of Ethereum accounts. Let's start with creating two accounts for Alice and Bob.

The following code snippet calls the account creation function of web3 library and creates two accounts.

```
var createAccounts = function () {
    var aliceKeys = web3.eth.accounts.create();
    console.log(aliceKeys);
    var bobKeys = web3.eth.accounts.create();
    console.log(bobKeys);
};
```

And here's the output that we get in the console window after running the previous snippet.

```
{
    address: '0xAff9d328E8181aE831Bc426347949EB7946A88DA',
    privateKey: '0x9fb71152b32cb90982f95e2b1bf2a5b6b2a5385
    5eacf59d132a2b7f043cfddf5',
    signTransaction: [Function: signTransaction],
    sign: [Function: sign],
    encrypt: [Function: encrypt]
}
{
    address: '0x22013fff98c2909bbFCcdABb411D3715fDB341eA',
    privateKey: '0xc6676b7262dab1a3a28a781c77110b63ab8cd5
    eae2a5a828ba3b1ad28e9f5a9b',
    signTransaction: [Function: signTransaction],
    sign: [Function: sign],
    encrypt: [Function: encrypt]
}
```

As you can see, along with the addresses and private keys, the output for each account creation function call also includes a few functions. For now, we will focus on the address and private key of the returned objects. The address is the Keccak-256 hash of the ECDSA public key of the generated private key. This address and private key combination represents an account on the Ethereum blockchain. You can send Ether to the address and you can spend that Ether using the private key of the corresponding address.

Get Test Ether in Sender's Account

Now, to create an Ethereum transaction which transfers Ether from one account to another, we first need some Ether in one of the accounts. Recall from the Bitcoin programming section that we used testnet faucets to get some test Bitcoins on the address we generated. We will do the same for Ethereum also. Remember that we are targeting the Ropsten test network for Ethereum, so we will search for a Ropsten faucet on the Internet. For this example, we submitted Alice's address that we generated in the previous code snippet to an Ethereum Ropsten test network faucet and we received three ethers on that address.

After receiving Ether on Alice's address, let's check the balance of this address to confirm if we really have the Ether or not. Though we can check the balance of this address using any of the Ethereum explorers online, let's do it using code. The following code snippet calls the getBalance function passing Alice's address as input parameter.

```
var getBalance = function () {
    web3.eth.getBalance('0xAff9d328E8181aE831Bc426347949
    EB7946A88DA').then(console.log);
};
```

And we get the following output as the balance of Alice's address. That's a huge number but that's actually the value of the balance in wei. Wei is the smallest unit of Ether. One Ether equals 10^18 wei. So, the following value equals three Ether, which is what we received from the test network faucet.

3000000000000000000

Prepare Ethereum Transaction

Now that we have some test Ether with Alice, let's create an Ethereum transaction to send some of this Ether to Bob. Recall that there are no inputs and outputs and UTXO queries to be done in the case of Ethereum because it uses an account and balances-based system. So, all that we need to do in the transaction is to specify the "from" address (the sender's address), the "to" address (the recipient address), and the amount of Ether to be sent, among a few other things.

Also, recall that in the case of a Bitcoin transaction we did not have to specify the transaction fee; however, in the case of an Ethereum transaction we need to specify two related fields. One is *gas* limit and the other is *gas* Price. Recall from Chapter 4 that *gas* is the unit of transaction fee we need to pay to the Ethereum network to get our transactions confirmed and added to blocks. *gas* Price is the amount of Ether (in gwei) we want to pay per unit of *gas*. The maximum fee that we allow to be used for a transaction is the product of *gas* and *gas* Price.

So, for this example transaction, we define a JSON object with the following fields. Here, "from" has Alice's address and "to" has Bob's address, and value is one Ether in wei. The *gas* Price we choose is 20 gwei and the maximum amount of *gas* we want to pay for this transaction is 42,000.

Also, note that we have left the data field empty. We will come back to this later in the smart contract section.

```
{
    from: "0xAff9d328E8181aE831Bc426347949EB7946A88DA",
    gasPrice: "20000000000",
    gas: "42000",
    to: '0x22013fff98c2909bbFCcdABb411D3715fDB341eA',
    value: "1000000000000000000",
    data: ""
}
```

Sign Transaction

Now that we have created a transaction object with the required fields and values, we need to sign it using the private key of the account that is sending the Ether. In this case, the sender is Alice, so we will use Alice's private key to sign the transaction. This is to cryptographically prove that it is actually Alice who is spending the Ether in her account.

```
var signTransaction = function () {
    var tx = {
        from: "0xAff9d328E8181aE831Bc426347949EB7946A88DA",
        gasPrice: "20000000000",
        gas: "42000",
        to: '0x22013fff98c2909bbFCcdABb411D3715fDB341eA',
        value: "1000000000000000000",
        data: ""
    };

    web3.eth.accounts.signTransaction(tx, '0x9fb71152b32cb
    90982f95e2b1bf2a5b6b2a53855eacf59d132a2b7f043cfddf5')
    .then(function(signedTx){
        console.log(signedTx.rawTransaction);
    });
};
```

The preceding code snippet calls the **`signTransaction`** function with the transaction object we created in the step before and Alice's private key that we got when we generated Alice's account. Following is the output we get when we run the prceding code snippet.

```
{
    messageHash: '0x91b345a38dc728dc06a43c49b92a6ac1e0e6d
    614c432a6dd37d809290a25aa6b',
    v: '0x2a',
    r: '0x14c20901a060834972a539d7b8ad1f23161
    c2144a2b66fbf567e37e963d64537',
    s: '0x3d2a0a818633a11832a5c48708a198af909
    eaf4884a7856c9ac9ed216d9b029c',
    rawTransaction: '0xf86c018504a817c80082a4109422013fff98c
    2909bbfccdabb411d3715fdb341ea880de0b6b3a76400
    00802aa014c20901a060834972a539d7b8ad1f23161c2144a2b66fbf5
    67e37e963d64537a03d2a0a818633a11832a5c48708a198af909ea
    f4884a7856c9ac9ed216d9b029c'
}
```

In the output of the **signTransaction** function we receive a JSON object with a few properties. The important value for us is the **rawTransaction** value. This is the hex string representation of the signed transaction. This is very similar to how we created a hex string of the Bitcoin transaction in the Bitcoin section.

Send Transaction to the Ethereum Network

The final step is to just send this signed raw transaction to the public-hosted Ethereum test network node, which we have set as the provider of our web3 object.

The following code calls the **sendSignedTransaction** function to send the raw transaction to the Ethereum test network. The input parameter is the value of the **rawTransaction** string that we got in the previous step as part of signing the transaction.

```
web3.eth.sendSignedTransaction(signedTx.rawTransaction).
then(console.log);
```

Notice the use of “then” in the prceding code snippet. This is interesting because the web3 library provides different levels of finality when working with Ethereum transactions, because an Ethereum transaction goes through several states after being submitted. In this function, call of sending a transaction to the network, then, is hit when the transaction receipt is created, and the transaction is complete.

After a few seconds, when the JavaScript promise resolves, the following is what we get as an output.

```
{
    blockHash: '0x26f1e1374d11d4524f692cdf1ce3aa6e085dcc1810
    8464229342 9eda3954d30e',
    blockNumber: 2514764,
    contractAddress: null,
    cumulativeGasUsed: 125030,
    from: '0xaff9d328e8181ae831bc426347949eb7946a88da',
    gasUsed: 21000,
    logs: [],
    logsBloom: '0x0000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    0000000000000',
    status: '0x1',
    to: '0x22013fff98c2909bbfccdabb411d3715fdb341ea',
    transactionHash: '0xd3f45394ac038c44c4fe6e0cdb7021fdbd
    672eb1abaa93eb6a1828df5edb6253',
    transactionIndex: 3
}
```

The output has a lot of information, as we can see. The most important part is the **transactionHash**, which is the ID of the transaction on the network. It also gives us the **blockHash**, which is the ID of the block in which this transaction was included. Along with this, we also get information about how much *gas* was used for this transaction, among other details. If the *gas* used is less than the maximum *gas* we specified during transaction creation, the remaining *gas* is sent back to the sender's address.

In this section, we sent a simple transaction to the Ethereum blockchain using JavaScript. But this is just the beginning of Ethereum application programming. In the next section, we will also look at how to create and call smart contracts programmatically.

Interacting Programmatically with Ethereum—Creating a Smart Contract

In this section, we will continue our Ethereum programming exercise, and we will create a simple smart contract on the Ethereum blockchain using the same web3 JavaScript library and the Infura service API.

Because, no computer programming beginners' tutorial is complete without a "Hello World" program, the smart contract we are going to create will be a simple smart contract returning the string "Hello World" when called.

The contract creation process will be a special kind of transaction sent to the Ethereum blockchain, and these types of transactions are called "contract creation transactions." These transactions do not mention a "to" address and the owner of the smart contract is the "from" address mentioned in the transaction.

Prerequisites

In this code exercise to create a smart contract, we will continue with the assumption that the web3 JavaScript library is installed and instantiated in a node.js app and we have registered for the Infura service, just like we did in the previous section.

Following are the steps to create a smart contract on Ethereum using JavaScript.

Program the Smart Contract

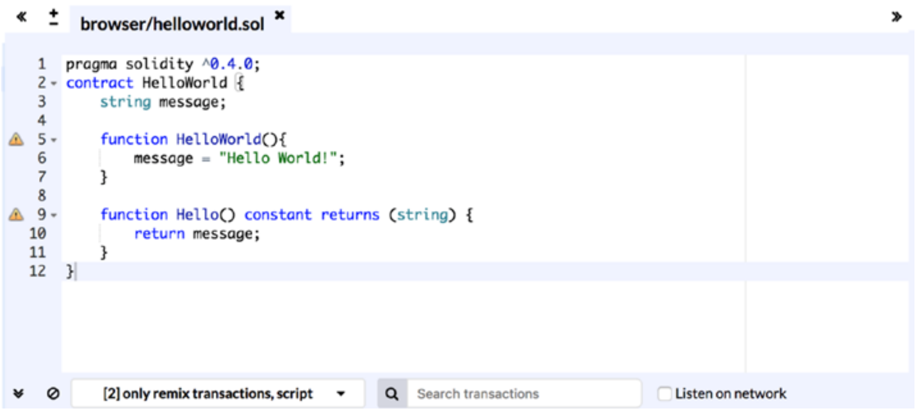
Recall from Chapter 4 that the Ethereum smart contracts are written in Solidity programming language. While the web3 JavaScript library will help us deploy our contract on the Ethereum blockchain, we will still have to write and compile our smart contract in Solidity before we send it to the Ethereum network using web3. So, let's first create a sample contract using Solidity.

There are a variety of tools available to code in Solidity. Most of the major IDEs and code editors have Solidity plugins for editing and compiling smart contracts. There is also a web-based Solidity editor called Remix. It's available for free to use at `https://remix.ethereum.org/`. Remix provides a very simple interface to code and compile smart contracts within your browser. In this exercise we will be using Remix to code and test our smart contract and then we will send the same contract to the Ethereum network using the web3 JavaScript library and the Infura API service.

The following code snippet is written in the Solidity programming language and it is a simple smart contract that returns the string "Hello World" from its function Hello. It also has a constructor that sets the value of the message returned.

```
pragma solidity ^0.4.0;
contract HelloWorld {
    string message;
    function HelloWorld(){
        message = "Hello World!";
    }
    function Hello() constant returns (string) {
        return message;
    }
}
```

Let's head to Remix and paste this code in the editor window. The following images (Figures 5-4 and 5-5) show how our sample smart contract looks in the Remix editor and what the output looks like when we clickeded the Create button on the right-side menu, under the Run tab. Also, note that by default, the Remix editor targets a JavaScript VM environment for smart contract compilation and it uses a test account with some ETH balance, for testing purposes. When we click the Create button, this contract is created using the selected account in the JavaScript VM environment.

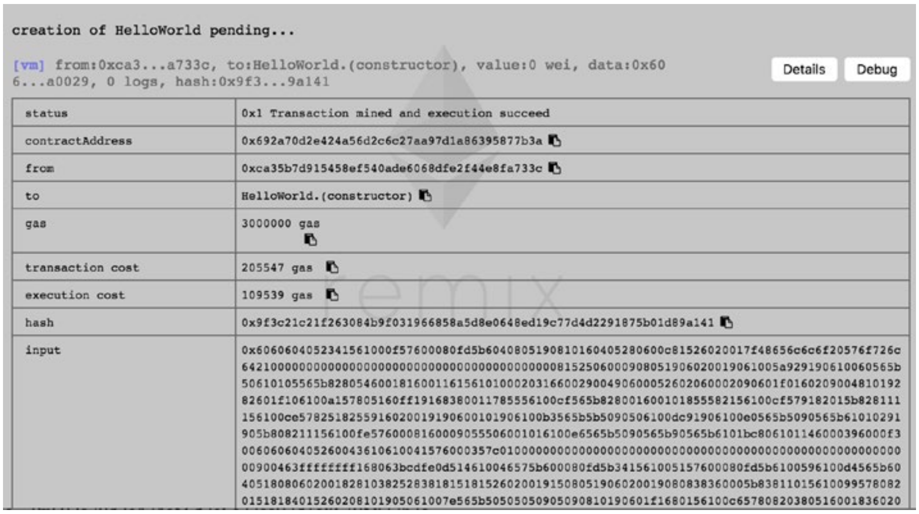


```
pragma solidity ^0.4.0;
contract HelloWorld {
    string message;

    function HelloWorld(){
        message = "Hello World!";
    }

    function Hello() constant returns (string) {
        return message;
    }
}
```

Figure 5-4. *Editing smart contracts in Remix IDE*



```
creation of HelloWorld pending...

[vm] from:0xca3...a733c, to:HelloWorld.(constructor), value:0 wei, data:0x60
6...a0029, 0 logs, hash:0x9f3...9a141
```

status	0x1 Transaction mined and execution succeed
contractAddress	0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	HelloWorld.(constructor)
gas	3000000 gas
transaction cost	205547 gas
execution cost	109539 gas
hash	0x9f3c21c21f263084b9f031966858a5d8e0648ed19c77d4d2291875b01d89a141
input	0x6060604052341561000f57600080fd5b6040805190810160405280600c81526020017f48656c6c6f20576f726c 64210000000000000000000000000000000000000000008152506000908051906020019061005a929190610060565b 50610105565b828054600181600116156101000203166002900490600052602060002090601f0160209004810192 82601f106100a157805160ff19168380011785555556100cf565b828001600101855558215 6100cf579182015b828111 156100ce578251825591602001919060010190610 0b3565b5b5090506100dc91906100e0565b5090565b61010291 905b808211156100fe5760008160009055506001016100e6565b5090565b90565b6101bc806101146000396000f3 00606060405260043610610041576000357c0100000000000000000000000000000000000000000000000000000000 00900463ffffffff168063bcdfe0d514610046575b600080fd5b341561005157600080fd5b6100596100d4565b60 40518080602001828103825283818151815260200191508051906020019080838360005b83811015610099578082 0151818401526020810190506100 7e565b50505050905090810190601f1680156100c657808203805160018360 20

Figure 5-5. *Smart contract creation output in Remix IDE*

Following is the output generated by the create operation, and it shows us that the contract has been created because it has a contract address. The “from” value is the account address that was used to create the contract. It also shows us the hash of the contract creation transaction.

```
status     0x1 Transaction mined and execution succeed
contractAddress    0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
from   0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to     HelloWorld.(constructor)
gas    3000000 gas
transaction cost   205547 gas
execution cost     109539 gas
hash   0x9f3c21c21f263084b9f031966858a5d8e0648ed19c77d4d2291
875b01d89a141
```

```
input  0x6060604052341561000f57600080fd5b6040805190810160405
280600c81526020017f48656c6c6f20576f726c642100000000000000000
00000000000000000000000815250600090805190602001906100 5a92919
0610060565b50610105565b82805460018160011615610100020316600290
0490600052602060002090601f016020900481019282601f106100a1578
05160ff19168380011785556100cf565b828001600101855582156100cf5
79182015b828111156100ce578251825591602001919060010190610 0b35
65b5b5090506100dc91906100e0565b5090565b61010291905b808211156
100fe576000816000905550600101610 0e6565b5090565b90565b6101bc8
06101146000396000f300606060405260043610610041576000357c01000
00000000000000000000000000000000000000000000000000000900463f
fffffff168063bcdfe0d514610046575b600080fd5b341561005157600080
fd5b6100596100d4565b604051808060200182810382528381815181526
02001915080519060200190808383600 05b83811015610099578082015180
840152602081019050610 07e565b50505050905090810190601f16801561
00c65780820380516001836020036101000a031916815260200191505b5
09250505060405180910390f35b6100dc61017c565b6000805460018160
0116156101000203166002900480601f01602080910402602001604051908
10160405280929190818152602001828054600181600116156101000203 1
660029004801561017257806 01f10610147576101008083540402835291 6
020019161017256 5b820191906000526020600020905b8154815290600 10
190602001808311610155578290 03601f168201915b50505050509050905
65b6020604051908101604052806000815250905600a165627a7a7230582
0d6796e48540eced3646ea52c632364666e64094479451066317789a712
aef4da0029
 decoded input  {}
 decoded output      -
 logs   []
 value  0 wei
```

At this point, we have a simple "*Hello World*" smart contract ready, and now the next step is to deploy it programmatically to the Ethereum blockchain.

Compile Contract and Get Details

Let's first get some details about our smart contract from Remix, which will be needed to deploy the contract to the Ethereum network using the web3 library. Click on the Compile tab in the right-side menu and then click the Details button. This pops up a new child window with details of the smart contract. What's important for us are the ABI and the BYTECODE sections on the details popup window.

Let's copy the details in the **`ABI`** section using the *copy value to clipboard* button available next to the ABI header. Following is the value of the ABI data for our smart contract.

```
[
    {
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [
            {
                "name": "",
                "type": "string"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }
]
```

This is a JSON array and if we closely look at it, we see that it has JSON objects for each function in our contract including its constructor. These JSON objects have details about a function and its input and output. This array describes the smart contract interface.

When we call this smart contract after it is deployed to the network, we will need this information to find out what functions the contract is exposing and what do we need to pass as an input to the function we wish to call.

Now let's get the data in the **BYTECODE** section of the details popup. Following is the data we copied for our contract.

```
{
    "linkReferences": {},
    "object": "60606040523415610000f57600080fd5b6040805190810
    160405280600c81526020017f48656c6c6f20576f726c6421000000
    0000000000000000000000000000000081525060009080519060200
    19061005a929190610060565b50610105565b828054600181600116
    1561010002031660029004906000526020600020906001f01602090048
    1019282601f106100a157805160ff1916838001178555556100cf565b8
    28001600101855582156100cf579182015b82811115610ce5782518
    2559160200191906001019061000b3565b5b5090506100dc91906100e
    0565b5090565b61010291905b808211156100fe576000816000905555
    06001016100e6565b5090565b90565b6101bc8061011460003960000f
    300606060405260043610610041576000357c0100000000000000000
    000000000000000000000000000000000000000900463ffffffff168
    063bcdfe0d514610046575b600080fd5b3415610051576000 80fd5b6
    100596100d4565b60405180806020018281038252838181518152602
    00191508051906020019080838360005b838110156100995780820 15
    181840152602081019050610 07e565b505050509050908101906 01f1
    680156100c6578082038051600183602003610 1000a031916815260 2
    00191505b509250505060405180910390f35b6100dc61017c565b600
    0805460018160011615610100020316600290048060 1f016020809 10
    402602001604051908101604052809291908181526020018280546 00
    18160011615610100020316600290048015610172578060 1f1061014
    757610100808354040283529160200191610172565b820191906000 5
    2602060002090 5b815481529060010190602001808311610155578 29
    003601f168201915b50505050509050905 65b6020604051908101604
    0528060008152509056 00a165627a7a72305820877a5da4f7e05c4ad
    9b45dd10fb6c133a523541ed06db6dd31d59b35d51768a30029",
```

```
"opcodes": "PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE
ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST
PUSH1 0x40 DUP1 MLOAD SWAP1 DUP2 ADD PUSH1 0x40 MSTORE
DUP1 PUSH1 0xC DUP2 MSTORE PUSH1 0x20 ADD PUSH32
0x48656C6C6F20576F726C64210000000000000000000000000000000000
000000 DUP2 MSTORE POP PUSH1 0x0 SWAP1 DUP1 MLOAD SWAP1 PUSH1
0x20 ADD SWAP1 PUSH2 0x5A SWAP3 SWAP2 SWAP1 PUSH2 0x60 JUMP
JUMPDEST POP PUSH2 0x105 JUMP JUMPDEST DUP3 DUP1 SLOAD PUSH1
0x1 DUP2 PUSH1 0x1 AND ISZERO PUSH2 0x100 MUL SUB AND PUSH1
0x2 SWAP1 DIV SWAP1 PUSH1 0x0 MSTORE PUSH1 0x20 PUSH1 0x0
KECCAK256 SWAP1 PUSH1 0x1F ADD PUSH1 0x20 SWAP1 DIV DUP2 ADD
SWAP3 DUP3 PUSH1 0x1F LT PUSH2 0xA1 JUMPI DUP1 MLOAD PUSH1 0xFF
NOT AND DUP4 DUP1 ADD OR DUP6 SSTORE PUSH2 0xCF JUMP JUMPDEST
DUP3 DUP1 ADD PUSH1 0x1 ADD DUP6 SSTORE DUP3 ISZERO PUSH2
0xCF JUMPI SWAP2 DUP3 ADD JUMPDEST DUP3 DUP2 GT ISZERO PUSH2
0xCE JUMPI DUP3 MLOAD DUP3 SSTORE SWAP2 PUSH1 0x20 ADD SWAP2
SWAP1 PUSH1 0x1 ADD SWAP1 PUSH2 0xB3 JUMP JUMPDEST JUMPDEST
```

```
POP SWAP1 POP PUSH2 0xDC SWAP2 SWAP1 PUSH2 0xE0 JUMP JUMPDEST
POP SWAP1 JUMP JUMPDEST PUSH2 0x102 SWAP2 SWAP1 JUMPDEST DUP1
DUP3 GT ISZERO PUSH2 0xFE JUMPI PUSH1 0x0 DUP2 PUSH1 0x0 SWAP1
SSTORE POP PUSH1 0x1 ADD PUSH2 0xE6 JUMP JUMPDEST POP SWAP1
JUMP JUMPDEST SWAP1 JUMP JUMPDEST PUSH2 0x1BC DUP1 PUSH2
0x114 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1 0x60
PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH2 0x41 JUMPI
PUSH1 0x0 CALLDATALOAD PUSH29 0x100000000000000000000000000
000000000000000000000000000000 SWAP1 DIV PUSH4 0xFFFFFFFF AND
DUP1 PUSH4 0xBCDFE0D5 EQ PUSH2 0x46 JUMPI JUMPDEST PUSH1 0x0
DUP1 REVERT JUMPDEST CALLVALUE ISZERO PUSH2 0x51 JUMPI PUSH1
0x0 DUP1 REVERT JUMPDEST PUSH2 0x59 PUSH2 0xD4 JUMP JUMPDEST
PUSH1 0x40 MLOAD DUP1 DUP1 PUSH1 0x20 ADD DUP3 DUP2 SUB DUP3
MSTORE DUP4 DUP2 DUP2 MLOAD DUP2 MSTORE PUSH1 0x20 ADD SWAP2
POP DUP1 MLOAD SWAP1 PUSH1 0x20 ADD SWAP1 DUP1 DUP4 DUP4 PUSH1
0x0 JUMPDEST DUP4 DUP2 LT ISZERO PUSH2 0x99 JUMPI DUP1 DUP3
ADD MLOAD DUP2 DUP5 ADD MSTORE PUSH1 0x20 DUP2 ADD SWAP1 POP
PUSH2 0x7E JUMP JUMPDEST POP POP POP POP SWAP1 POP SWAP1 DUP2
ADD SWAP1 PUSH1 0x1F AND DUP1 ISZERO PUSH2 0xC6 JUMPI DUP1
DUP3 SUB DUP1 MLOAD PUSH1 0x1 DUP4 PUSH1 0x20 SUB PUSH2 0x100
EXP SUB NOT AND DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP JUMPDEST
POP SWAP3 POP POP POP PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1
RETURN JUMPDEST PUSH2 0xDC PUSH2 0x17C JUMP JUMPDEST PUSH1 0x0
DUP1 SLOAD PUSH1 0x1 DUP2 PUSH1 0x1 AND ISZERO PUSH2 0x100
MUL SUB AND PUSH1 0x2 SWAP1 DIV DUP1 PUSH1 0x1F ADD PUSH1
0x20 DUP1 SWAP2 DIV MUL PUSH1 0x20 ADD PUSH1 0x40 MLOAD SWAP1
DUP2 ADD PUSH1 0x40 MSTORE DUP1 SWAP3 SWAP2 SWAP1 DUP2 DUP2
MSTORE PUSH1 0x20 ADD DUP3 DUP1 SLOAD PUSH1 0x1 DUP2 PUSH1 0x1
AND ISZERO PUSH2 0x100 MUL SUB AND PUSH1 0x2 SWAP1 DIV DUP1
ISZERO PUSH2 0x172 JUMPI DUP1 PUSH1 0x1F LT PUSH2 0x147 JUMPI
```

```
PUSH2 0x100 DUP1 DUP4 SLOAD DIV MUL DUP4 MSTORE SWAP2 PUSH1
0x20 ADD SWAP2 PUSH2 0x172 JUMP JUMPDEST DUP3 ADD SWAP2 SWAP1
PUSH1 0x0 MSTORE PUSH1 0x20 PUSH1 0x0 KECCAK256 SWAP1 JUMPDEST
DUP2 SLOAD DUP2 MSTORE SWAP1 PUSH1 0x1 ADD SWAP1 PUSH1 0x20
ADD DUP1 DUP4 GT PUSH2 0x155 JUMPI DUP3 SWAP1 SUB PUSH1 0x1F
AND DUP3 ADD SWAP2 JUMPDEST POP POP POP POP POP SWAP1 POP
SWAP1 JUMP JUMPDEST PUSH1 0x20 PUSH1 0x40 MLOAD SWAP1 DUP2
ADD PUSH1 0x40 MSTORE DUP1 PUSH1 0x0 DUP2 MSTORE POP SWAP1
JUMP STOP LOG1 PUSH6 0x627A7A723058 KECCAK256 DUP8 PUSH27
0x5DA4F7E05C4AD9B45DD10FB6C133A523541ED0
6DB6DD31D59B35D5 OR PUSH9 0xA3002900000000000 ",
    "sourceMap": "24:199:0:-;;;75:62;;;;;;;;106:24;;;;;;;;;;;
    ;;;;;;:7;:24;;;;;;;;;;;;;:::i;:::-;;24:199;;;;;;;;;;;;;;;
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;;;;;;;;;;;;;;;;;;;;;;;:::i;:::-;;;;::o;:::-;;;;;;;;;;;;;;;
    ;;;;;;;;;;;;:::o;:::-;;;;;;;"
}
```

As we can see, the data in the BYTECODE section is a JSON object. This is basically the output of the compilation of the smart contract. Remix compiled our smart contract using the Solidity compiler and as a result we got the solidity byte code. Now closely examine this JSON and look at the "object" property and its value. This is a hex string that contains the byte code for our smart contract, and we will be sending it in the contract creation transaction in the data field–the same data field that we left blank in the previous example Ethereum transaction between Alice and Bob.

Now we have all the details for our smart contract and we are ready to send it to the Ethereum network.

Deploy Contract to Ethereum Network

Now that we have our smart contract and its details, we need to prepare a transaction that can deploy this contract to the Ethereum blockchain. This transaction preparation will be very similar to the transaction we prepared in the previous section, but it will have a few more properties that are needed to create contracts.

First, we need to create an object of the **`web3.eth.Contract`** class, which can represent our contract. The following code snippet creates an instance for the said class with a JSON array as an input parameter. This is the same JSON array that we copied from the ABI section of the Remix popup window, showing the details about our smart contract.

```
var helloworldContract = new web3.eth.Contract([{
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }, {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }]);
```

Now we need to send this contract to the Ethereum network using the **`Contract.deploy`** method of the web3 library. The following code snippet shows how to do this.

```
helloworldContract
.deploy({
        data: '0x6060604052341561000f57600080fd5b604080519081
        0160405280600c81526020017f48656c6c6f20576f726c6421000
        00000000000000000000000000000000000008152506000908051
        906020019061005a929190610060565b50610105565b828054600
        18160011615610100020316600290049060005260206000209060
        1f016020900481019282601f106100a157805160ff19168380011
        785556100cf565b828001600101855582156100cf579182015b82
        8111156100ce5782518255916020019190600101906100b3565b5
        b5090506100dc91906100e0565b5090565b61010291905b808211
        156100fe5760008160009055506001016100e6565b5090565b905
        65b6101bc806101146000396000f3006060604052600436106100
        41576000357c01000000000000000000000000000000000000000
        0000000000000000900463ffffffff168063bcdfe0d514610046
        575b600080fd5b341561005157600080fd5b6100596100d4565b6
        04051808060200182810382528381815181526020019150805190
        6020019080838360005b838110156100995780820151818401526
        0208101905061007e565b50505050905090810190601f16801561
        00c65780820380516001836020036101000a03191681526020019
        1505b509250505060405180910390f35b6100dc61017c565b6000
        8054600181600116156101000203166002900480601f016020809
        10402602001604051908101604052809291908181526020018280
        54600181600116156101000203166002900480156101725780601
        f1061014757610100808354040283529160200191610172565b82
        0191906000526020600020905b815481529060010190602001808
        3116101555782900360 1f168201915b50505050509050 90565b60
        2060405190810160405280600081525090 5600a165627a7a72305
        820877a5da4f7e05c4ad9b45dd10fb6c133a523541ed06db6dd31
        d59b35d51768a30029'
    })
    .send({
        from: '0xAff9d328E8181aE831Bc426347949EB7946A88DA',
        gas: 4700000,
        gasPrice: '20000000000000'
    },
    function(error, transactionHash){
        console.log(error);
        console.log(transactionHash);
    })
    .then(function(contract){
        console.log(contract);
    });
```

Note that the value of the field data inside the deploy function parameter object is the same value we received in the object field of the BYTECODE details in the previous step. Also notice that the string "0x" is added to this value in the beginning. So, the data passed in the deploy function is '0x' + byte code of the contract.

Inside the send function after the deploy, we have added the "from" address, which will be the owner of the contract and the transaction fee details of *gas* limit and *gas* Price. Finally, when the call is complete, the contract object is returned. This contract object will have the contract details along with the address of the contract, which can be used to call the function on the contract.

Another way of sending the contract to the network would be to wrap the contract inside a transaction and send it directly. The following code snippet creates a transaction object with data as the contract bytecode, signs it using the private key of the address in the “from” field, and then sends it to the Ethereum blockchain.

Note that we have not assigned a “to” address in this transaction object, as the address of the contract is unknown before the contract is deployed.

```
var tx = {
        from: "0x22013fff98c2909bbFCcdABb411D3715fDB341eA",
        gasPrice: "20000000000",
        gas: "4900000",
        data: "0x6060604052341561000f57600080fd5b604080519081
        0160405280600c81526020017f48656c6c6f20576f726c6421000
        00000000000000000000000000000000000008152506000908051
        906020019061005a929190610060565b50610105565b828054600
        181600116156101000203166002900490600052602060002090 60
        1f016020900481019282601f106100a157805160ff19168380011
        785556100cf565b828001600101855582156100cf579182015b82
        8111156100ce578251825591602001919060010190610 0b3565b5
        b5090506100dc91906100e0565b5090565b61010291905b808211
        156100fe5760008160009055506001016100e6565b5090565b905
        65b6101bc806101146000396000f300606060405260043610610 0
        41576000357c01000000000000000000000000000000000000000
        00000000000000000900463ffffffff168063bcdfe0d514610046
        575b600080fd5b341561005157600080fd5b6100596100d4565b6
        040518080602001828103825283818151815260200191508051 90
        602001908083836000 5b83811015610099578082015181840152 6
        0208101905061007e565b50505050905090810190601f16801561
        00c65780820380516001836020036101000a031916815260200 19
        1505b509250505060405180910390f35b6100dc61017c565b6000
        80546001816001161561010002031660029004806 01f016020809
        10402602001604051908101604052809291908181526020018280
        546001816001161561010002031660029004801561017257806 01
        f1061014757610100808354040283529160200191610172565b82
        0191906000526020600020905b815481529060010190602001808
        311610155578290036001f168201915b50505050509050909565b60
        2060405190810160405280600081525090560 0a165627a7a72305
        820877a5da4f7e05c4ad9b45dd10fb6c133a523541ed06db6dd31
        d59b35d51768a30029"
    };

    web3.eth.accounts.signTransaction(tx, '0xc6676b7262dab1a3
    a28a781c77110b63ab8cd5eae2a5a828ba3b1ad28e9f5a9b')
    .then(function (signedTx) {
        web3.eth.sendSignedTransaction(signedTx.rawTransaction)
        .then(console.log);
    });
```

When we execute this code snippet, we get the following output, which is the receipt of this transaction.

```
{
    blockHash: '0xaba93b4561fc35e062a1ad72460e0b677603331bbee
    3379ce6c74fa5cf505d82',
    blockNumber: 2539889,
    contractAddress: '0xd5a2d13723A34522EF79bE0f1E7806E86a45
    78E9',
    cumulativeGasUsed: 205547,
    from: '0x22013fff98c2909bbfccdabb411d3715fdb341ea',
    gasUsed: 205547,
    logs: [],
    logsBloom: '0x0000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000
    0000000000000',
    status: '0x1',
    to: null,
    transactionHash: '0xc333cbc5fc93b52871689aab22c48b910cb19
    2b4875bea69212363030d36565a',
    transactionIndex: 0
}
```

Notice the properties of the transaction receipt object. It has a value assigned to the **`contractAddress`** property, while the value of the "to" property is null. This means that this was a contract creation transaction that was successfully mined on the network and the contract created as part of this transaction is deployed at the address `**0xd5a2d13723A34522EF79bE0f1E7806E86a4578E9**`.

We have successfully created an Ethereum smart contract programmatically.

Interacting Programmatically with Ethereum—Executing Smart Contract Functions

Now that we have deployed our smart contract to the Ethereum network, we can call its member functions. Following are the steps to call an Ethereum smart contract programmatically.

Get Reference to the Smart Contract

To execute a function of the smart contract, first we need to create an instance of the web3.eth.Contract class with the ABI and address of our deployed contract. The following code snippet shows how to do that.

```
var helloworldContract = new web3.eth.Contract([{
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }, {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }], '0xd5a2d13723A34522EF79bE0f1E7806E86a4578E9');
```

In the prceding code snippet, we have created an instance of the **web3.eth.Contract** class by passing the ABI of the contract we created in the previous section, and we have also passed the address of the contract that we received after deploying the contract.

This object can now be used to call functions on our contract.

Execute Smart Contract Function

Recall that we have only one public function in our contract. This method is named Hello and it returns the string **`"Hello World!"`** when executed.

To execute this method, we will call it using the **`contract.methods`** class in the web3 library. The follwing code snippet shows this.

```
helloworldContract.methods.Hello().send({
        from: '0xF68b93AE6120aF1e2311b30055976d62D7dBf531'
    }).then(console.log);
```

In the prceding code snippet, we have added a value to the “from” address in the send function, and this address will be used to send the transaction that will in turn execute the function Hello on our smart contract.

The full code for calling a smart contract is in the follwing code snippet.

```
var callContract = function () {
    var helloworldContract = new web3.eth.Contract([{
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }, {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }], '0xd5a2d13723A34522EF79bE0f1E7806E86a4578E9');
    helloworldContract.methods.Hello().send({
        from: '0xF68b93AE6120aF1e2311b30055976d62D7dBf531'
    }).then(console.log);
};
```

Another way of executing this contract function will be by sending a raw transaction by signing it. It is similar to how we sent a raw Ethereum transaction to send Ether and to create a contract in the previous sections. In this case all we need to do is provide the contract address in the "to" field of the transaction object and the encoded ABI value of the function call in the data field.

The following code snippet first creates a contract object and then gets the encoded ABI value of the smart contract function to be called. It then creates a transaction object based on these values and then signs and sends it to the network. Note that we have used the **encodeABI** function on the contract function to get the data payload value for the transaction. This is the input for the smart contract.

```
var callContract = function () {
    var helloworldContract = new web3.eth.Contract([{
        "constant": true,
        "inputs": [],
        "name": "Hello",
        "outputs": [{
            "name": "",
            "type": "string"
        }],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }, {
        "inputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "constructor"
    }], '0xd5a2d13723A34522EF79bE0f1E7806E86a4578E9');

    var payload = helloworldContract.methods.Hello().
    encodeABI();

    var tx = {
        from: "0xF68b93AE6120aF1e2311b30055976d62D7dBf531",
        gasPrice: "20000000000",
        gas: "4700000",
        data: payload
    };

    web3.eth.accounts.signTransaction(tx, '0xc6676b7262dab1a3
    a28a781c77110b63ab8cd5eae2a5a828ba3b1ad28e9f5a9b')
        .then(function (signedTx) {
            web3.eth.sendSignedTransaction(signedTx.raw
            Transaction)
            .then(console.log);
    });
};
```

Important Note When using a public-hosted node for Ethereum, we should use the raw transaction method for creating and executing smart contracts because the web3.eth.Contract submodule of the library uses either an unlocked or default account associated with the provider Ethereum node, but this is not supported by the public nodes (at the time of this writing).

Blockchain Concepts Revisited

In the previous sections we programmatically sent transactions to both Bitcoin and Ethereum blockchains using JavaScript. Here are some of the common concepts that we can now revisit, looking at the process of handcrafting transactions using code.

- **Transactions**: Looking at the code we wrote and the output we got for sending transactions to Ethereum and Bitcoin, we can now say that blockchain transactions are the operations initiated from an account owner, which, if completed successfully, update the state of the blockchain. For example, in our transactions between Alice and Bob, we saw that the ownership of a certain amount of Bitcoins and Ether changed from Alice to Bob and vice versa, and this change of ownership was recorded in the blockchain, hence bringing it into a new state. In the case of Ethereum, transactions go further into contract creation and execution and these transactions also update the state of the blockchain. We created a transaction that in turn deployed a smart contract on the Ethereum blockchain. The state of the blockchain was updated because now we have a new contract account created in the blockchain.
- **Inputs, Outputs, Accounts and Balances**: We also saw how Bitcoin and Ethereum are different from each other in terms of managing the state. While Bitcoin uses the UTXO model, Ethereum uses the accounts and balances model. However, the underlying idea is both the blockchains record the ownership of assets, and transactions are used to change ownership of these assets.

- **Transaction Fee**: For every transaction we do on public blockchain networks, we must pay a transaction fee for our transactions to be confirmed by the miners. In Bitcoin this is automatically calculated, while in Ethereum we should mention the maximum fee we are willing to pay in terms of *gas* Price and *gas* limit.
- **Signing**: In both cases, we also saw that after creating a transaction object with the required values, we signed it using the sender's public key. Cryptographic signing is a way of proving ownership of the assets. If the signature is incorrect, then the transaction becomes invalid.
- **Transaction broadcasting**: After creating and signing the transactions, we sent them to the blockchain nodes. While we sent our example transactions to publicly hosted Bitcoin and Ethereum test network nodes, we are free to send our transactions to multiple nodes if we don't trust all of them to process our transactions. This is called transaction broadcasting.

To summarize, when interacting with blockchains, if we intend to update the state of the blockchain, we submit signed transactions; and to get these transactions confirmed, we need to pay some fee to the network.

Public vs. Private Blockchains

Based on access control, blockchains can be classified as public and private. Public blockchains are also called *permissionless* blockchain and private blockchains are also called *permissioned* blockchains. The primary difference between the two is access control. Public or permissionless blockchains do not restrict addition of new nodes to the network and anyone can join the network. Private blockchains have a limited number of nodes in the network and not everyone can join the network. Examples of public blockchains are Bitcoin and Ethereum main nets. An example of a private blockchain can be a network of a few Ethereum nodes connected to each other but not connected to the main net. These nodes would be collectively called a private blockchain.

Private blockchains are generally used by enterprises to exchange data among themselves and their partners and/or among their suborganizations.

When we develop applications for blockchains, the type of blockchain, public or private, makes a difference because the rules of interaction with the blockchain may or may not be the same. This is called blockchain governance. The public blockchains have a predefined set of rules and the private ones can have a different set of rules per blockchain. A private blockchain for a supply chain may have different governance rules, while a private blockchain for protocol governance may have different rules. For example, the token, *gas* Price, transaction fee, endpoints, etc. may or may not be the same in the aforementioned private Ethereum ledger and the Ethereum main net. This can impact our applications too.

In our code samples, we primarily focused on the public test networks of Bitcoin and Ethereum. While the basic concepts of interacting with private deployments of these blockchains will still be the same, there will be differences in how we configure our code to point to the private networks.

Decentralized Application Architecture

In general, the decentralized applications are meant to directly interact with the blockchain nodes without the need for any centralized components coming into picture. However, in practical scenarios, with legacy systems integrations and limited functionality and scaling of the current blockchain networks, sometimes we must make choices between full decentralization and scalability while designing our DApps.

Public Nodes vs. Self-Hosted Nodes

Blockchains are decentralized networks of nodes. All nodes have the same copy of data and they agree on the state of data always. When we develop applications for blockchains, we can make our application talk to any of the nodes of the target network. There can be mainly two set-ups for this:

- **Application and node both run locally:** The application and the node both run on the local machine. This means we will need our application users to run a local blockchain node and point the application to connect with it. This model would be a purely decentralized model of running an application. An example of this model is the Ethereum-based Mist browser, which uses a local *geth* node.

  Figure 5-6 shows this setup.

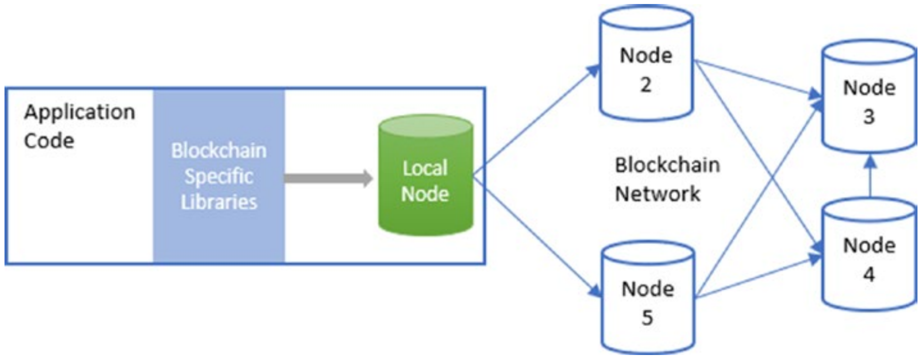


Figure 5-6. *DApp connets to local node*

- **Public node:** The application talks to a public node hosted by a third party. This way our users don't have to host a local node. There are several advantages and disadvantages of this approach. While the users don't have to pay for power and storage for running a local node, they need to trust a third party to broadcast their transactions to the blockchain. The Ethereum browser plugin metamask uses this model and connects with public hosted Ethereum nodes.

Figure 5-7 shows this setup.

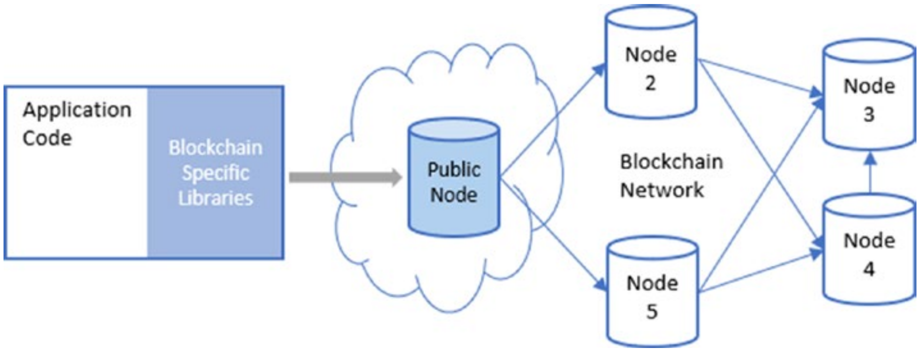


Figure 5-7. *DApp connets to public node*

Decentralized Applications and Servers

Apart from the previously mentioned scenarios, there can be other setups too, depending upon specific use cases and requirements. There are a lot of scenarios when a server is needed between an app and the blockchain. For example: When you need to maintain a cache of the blockchain state for faster queries; when the app needs to send notifications (emails, push, SMS, etc.) to the users based on state updates on the blockchain; and when multiple ledgers are involved, and you need to run a back-end logic to transform data between the ledgers. Imagine the infrastructure being used by some of the big cryptocurrency exchanges where we get all the services like two-factor authentication, notifications, and payment gateways, among other things, and none of these services are available directly in any of the blockchains. In a broader sense, blockchains simply make sure of keeping the data layer tamper resistant and auditable.

Summary

In this chapter we learned about decentralized application development along with some code exercises about interacting programmatically with the Bitcoin and Ethereum blockchains. We also looked at some of the DApp architecture models and how they differ based on the use cases.

In the next chapter we will set up a private Ethereum network and then we will develop a full-fledged DApp interacting with this private network, which will also use smart contracts for business logic.

References

web3.js Documentation

http://web3js.readthedocs.io/en/1.0/index.html.

Solidity Documentation

https://solidity.readthedocs.org/.

bitcoinjs Source Code Repository

https://github.com/bitcoinjs/bitcoinjs-lib.

Infura Documentation

https://infura.io/docs.

Block Explorer API Documentation

https://blockexplorer.com/api-ref.

Designing the Architecture for your Ethereum Application

https://blog.zeppelin.solutions/designing-the-architecture-for-your-ethereum-application-9cec086f8317.