ICT3009 Assignment: Blockchain & Smart Contracts I

Quentin Falzon, CS

February 2021

Contents

1 Introduction

This assignment aims to provide the functionality for a decentralized loan platform, available to users on the same peer-to-peer (P2P) network. Such a lending system provides two main advantages over traditional centralized systems i.e., the bank.

- **It lowers the threshold required for being granted a loan.** The bank will typically require extensive information about a person's identity and legal conduct, credit history, collateral and income. Based on these factors, they may or may not provide a loan. On a decentralized network however, there is no hard policy for every loan. Rather, it is up to each user on the platform to decide whether they shall provide a loan or not.
- **It offers high-risk, high-reward investment opportunities.** Users requesting loans on the P2P network are able to specify how much interest they are willing to pay on a loan. Depending on their financial situation and the urgency of the loan, the offered interest may vary significantly from between users. This provides good short or long-term investment opportunities for users seeking to fulfill loan requests which fit their terms.

For a loan to take place, there are three **different** parties (addresses) which need to be associated with a *LoanRequest*.

1. **The Borrower.** The issuer of a *LoanRequest* is the borrower, as their wallet address becomes associated with the request. They specify an amount of interest i they are willing to pay on the loan. When a guarantor places a guarantee on their request, the borrower must accept or reject the guarantee. Only then can a lender provide the loan.
2. **The Guarantor.** This user on the P2P network sends funds into the smart contract which guarantee that any loan will be honoured to the lender, should the borrower not pay back in time or at all. This means it is possible for a guarantor to lose their funds, since the guarantee itself is not guaranteed. In view of this, the guarantor is assumed to have collateral over the borrower. The guarantor specifies how much of the interest they want, g.
3. **The Lender.** The user providing the loan to the borrower. A lender is not able to command an amount of interest for themselves, since it is fixed at $(i - g)$. Instead, they can browse through all *LoanRequests*, and select one or many with the most attractive terms (payback period and lender interest).

2 LoanPlatform.sol

2.1 Implementation

This smart contract implements the required functionality as specified in problem 1. Once deployed, it acts as a platform through which loans can be requested, guaranteed, provided, paid back etc. The core functions and modifiers used in the solution are outlined below.

2.1.1 struct LoanRequest

My implementation is centred around this struct. Once submitted by a borrower, a *LoanRequest* is permanent within the smart contract. Its fields are updated to reflect the request's current status. The smart contract keeps a global *requests* array of the structs, which is not public. Each struct is defined by the following variables:

- *address payable borrower*
- *address payable guarantor*
- *address payable lender*
- *uint256 sum*
- *uint256 interest*
- *uint256 paybackPeriod*
- *uint256 guarantorInterest*
- *uint256 lenderInterest*
- *uint256 status*

Where *status* is:
0 → has no accepted guarantee.
1 → has an accepted guarantee.
2 → has been provided a loan.
3 → has been payed back by borrower.
4 → has not been paid back by borrower, and guarantee was claimed by lender.

2.1.2 function _indexInRange(index)

Returns true if the index is in range of the *requests* array. Otherwise, returns false.

2.1.3 modifier indexInRange(index)

Requires that **_indexInRange(index)** returns true. Otherwise, reverts the transaction. Useful for preventing index-out-of-bounds errors.

2.1.4 modifier isBorrower(index)

Requires that the message sender is the borrower associated with the *LoanRequest* at the given index. Useful for ensuring a borrower is unable to provide a guarantee or loan to their own *LoanRequest*.

2.1.5 function submitRequest(sum, interest, paybackPeriod)

Pushes a new *LoanRequest* struct onto the *requests* array. Does not allow sum to be zero, but zero interest is allowed. Sets guarantor and lender addresses to the **zero address**. Sets borrower address to the address of the function caller. This function is public, so it can be called by anyone on the network.

2.1.6 function viewRequest(index)

Uses the modifier in 2.1.3. Returns all fields of the *LoanRequest* at $requests[index]$.

2.1.7 function guarantee(index, gInterest)

Uses the modifier in 2.1.3. This function allows a guarantor to place a guarantee on a request. This function is **payable**, meaning it accepts funds to be sent into the smart contract. Require conditions are set up to ensure that:

1. The funds passed in are exactly equal to the loan value.
2. The function caller is not the borrower.
3. There is no pending guarantee i.e., the guarantor is currently the zero address.
4. The guarantor interest is less than the interest offered to pay by the borrower.
5. The *LoanRequest* is not expired.

The function is payable, so funds can be transferred every time it is called. However, the smart contract should prevent a guarantor from providing a guarantee more than once. This is achieved by setting the guarantor address to the function caller's address. Now, if the guarantor were to accidentally re-run **guarantee()**, the 5^{th} require condition would fire, thereby reverting the transaction.

2.1.8 function lend(index)

Uses the modifier in 2.1.3. This function allows a lender to provide a loan for a *LoanRequest*. The function is also **payable**. Require conditions are set up to ensure that:

1. The funds passed in are exactly equal to the loan value.
2. The function caller is neither the guarantor nor the borrower.
3. The *LoanRequest* status is 1, meaning it has an accepted guarantee, but has not yet been provided a loan.
4. The *LoanRequest* is not expired.

Similarly to **guarantee()**, the smart contract should prevent a lender from calling **lend()** multiple times. This is achieved by manipulating *LoanRequest.status*. Figure 8 illustrates the how *status* changes through a request's lifecyle.

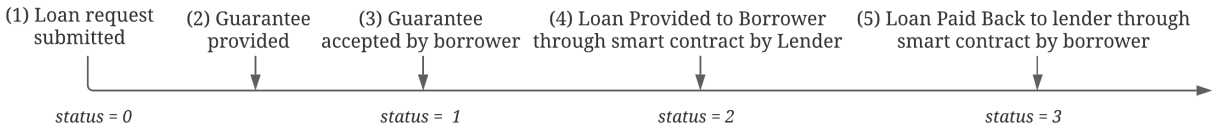

Figure 1: A healthy *LoanRequest* lifecycle

After the 4 require conditions pass, *status* is immediately set to 2. Next, the sender address is associated with the lender field, and *LoanRequest.lenderInterest* is computed. Finally, the loan is transferred from the smart contract to the borrower. The transfer is transacted at the end of the function in the interest of preventing **reentrancy attacks**. Any reenctrant call to **lend()** will cause the 3^{rd} require condition to fire, thereby reverting the transaction.

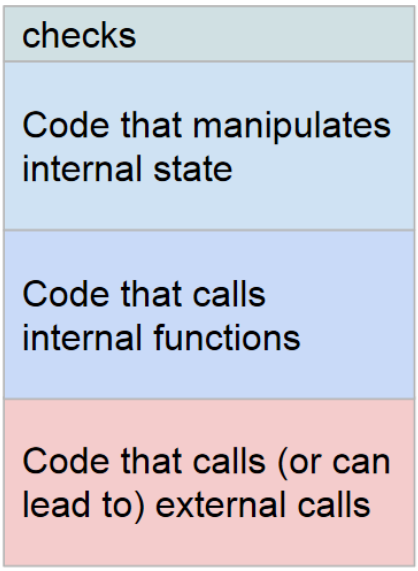

Figure 2: Best practice for structuring functions to avoid reentrancy bugs

2.1.9 function accept(index)

Uses the modifiers in 2.1.3 and 2.1.4. Requires that *LoanRequest.status* is 0, then sets it to 1.

2.1.10 function reject(index)

Uses the modifiers in 2.1.3 and 2.1.4. This function can be called by a borrower to reject the guarantee, even after they have accepted it, but before a lender has provided a loan. Therefore, the smart contract caters for the eventuality where the borrower changes their mind about wanting a loan, or decides to resubmit a smaller/larger loan request. Require conditions are set up to ensure that:

1. The borrower cannot reject the guarantee if a loan has already been provided or paid back i.e., *LoanRequest.status* is not 2 or 3.
2. There is a guarantee to reject i.e., *LoanRequest.status* is not 0.

This function is not payable, however it still involves funds being sent from the smart contract back to the guarantor. Therefore, *LoanRequest.status* is set to 0 immediately after the require conditions pass, preventing reentrancy. The **transfer()** is also executed as late as possible throughout the function.

2.1.11 function payBack(index)

Uses the modifiers in 2.1.3 and 2.1.4. This function allows a borrower to pay back the loan, including guarantor and lender interest. Require conditions are set up to ensure that:

1. The funds passed in are exactly equal to (*sum + guarantorInterest + lenderInterest*).
2. The *LoanRequest* has a loan provided.
3. The loan has not already been paid back.

With reference to figure 8, *status* is immediately updated to 3, preventing reentrant calls. Finally, (*sum + guarantorInterest*) is transferred from within the smart contract to the guarantor, and (*sum + lenderInterest*) is transferred from within the smart contract to the lender.

2.1.12 missedPayBack(index)

Uses the modifier in 2.1.3. This function is called by the lender when a loan has expired and the borrower has not yet paid it back. Require conditions are set up to ensure that:

1. The lender has not yet been paid back by the borrower, or claimed the guarantee already.
2. The loan has expired.
3. Only the lender can claim the guarantee.

Following the passing of all require conditions, *status* is immediately updated to 4, preventing reentrant calls. Finally, the original loan value (excluding any interest) is transferred from the smart contract to the lender.

2.2 Design Patterns

- **Access Restriction:** The modifiers in 2.1.3 and 2.1.4 are used as generally applicable modifiers which ensure requirements are met prior to the execution of a function. For instance, only a borrower should be able to call **payback()**, therefore the **isBorrower()** modifier is applied before execution.
- **State Machine:** The smart contract goes through several behavioural stages, where some functionality should be disallowed. *LoanRequest.status* is used as a state machine to ensure correct behaviour throughout.
- **Pull Payment:** The **missedPayBack()** function adheres to the *pull payment* design pattern. When a smart contract sends funds to another party, it is possible for the transaction to fail. Instead, it is the receiver's (lender's) responsibility to withdraw their funds in the event of a missed payment.
- The **Automatic Depreceation Pattern** could have been implemented by writing a modifier **loanIsNotExpired()**, however this check was needed at most twice throughout the entire contract, so require conditions were used directly within the respective functions **guarantee()** and **lend()**.

3 LoanToken.sol

3.1 The ERC20 Token

The Ethereum blockchain supports the creation and use of **tokens**, which can represent cryptocurrencies, company shares, loyalty points etc. Tokens are created on the blockchain by a smart contract. The smart contract should be responsible for:

- Creating tokens
- Handling token transactions correctly
- Keeping track of token holder balances

Of course, there are many ways to go about implementing the above functionality. However, having many kinds of tokens defined in different ways is undesirable. Suppose the various tokens are to be made available on an exchange platform. Translation code must be written as an interface between the exchange platform and each uniquely-defined token smart contract. This can be a very time consuming taks to undertake, and is caused by the lack of standardization between each token smart contract.

ERC20 solves this problem by defining a standard for token creation. There are six mandatory functions, namely:

- **totalSupply()**. Returns the total token supply.
- **balanceOf(_owner)**. Returns the token balance of an account.
- **transfer(_to, _value)**. Transfers an amount of tokens to a specified address, and throws an error if there are insufficient funds.
- **transferFrom(_from, _to, _value)**. Similar to **transfer()**, but allows the specification of a sender address.
- **approve(_spender, _value)**. Allows spender to withdraw from the caller's account multiple times, up to the specified value amount.
- **allowance(_owner, _spender)**. Returns the amount which a spender is allowed to withdraw from the contract owner.

ERC20 also specifies three optional functions:

- **name()**. Returns the token name e.g., 'LoanToken'.
- **symbol()**. Returns the token symbol e.g, 'LOAN'.
- **decimals()**. Returns the number of decimals the token uses for its representation.

3.2 Implementation

The functionality outlined above was implemented in the smart contract LoanToken.sol, with the exception of the optional **decimals()** function. It is worth noting that solidity automatically generates getter functions for public variables, so **totalSupply()**, **balanceOf()** and **allowance()** were simply declared as public variables.

The smart contract was used in truffle develop to perform some basic transactions and ensure it is working as expected. This can also be achieved more conveniently within Remix[1].

```
truffle(develop)> web3.eth.getAccounts().then(function(acc){ accounts = acc })
undefined
truffle(develop)> quentin = accounts[0]
'0xfdb257532A5b7D84B969a2A9ECAF3Ceca258f22a'
truffle(develop)> sweetShop = accounts[1]
'0x99073652c27bEaDEFa2F7810E0908A78339c675d'
truffle(develop)> alice = accounts[2]
'0x65600B30852B45E6780a2fA986dD941155AdA80c'
truffle(develop)> LoanToken.deployed().then(function(instance) { tokenInstance = instance; })
undefined
truffle(develop)> tokenInstance.name()
'LoanToken'
truffle(develop)> tokenInstance.symbol()
'LOAN'
truffle(develop)> tokenInstance.totalSupply().then(function(s) { supply = s; })
undefined
truffle(develop)> supply.toNumber()
1000000
```

Figure 3: Deploying LoanToken.sol

```
truffle(develop)> let balQ = await tokenInstance.balanceOf(quentin)
undefined
truffle(develop)> balQ.toNumber()
1000000
truffle(develop)> let balS = await tokenInstance.balanceOf(sweetShop)
undefined
truffle(develop)> balS.toNumber()
0
truffle(develop)> let balA = await tokenInstance.balanceOf(alice)
undefined
truffle(develop)> balA.toNumber()
0
truffle(develop)>
```

Figure 4: Checking initial account balances. LoanToken allocates an initialSupply of 1 million LoanTokens to the smart contract owner.

[1] http://remix.ethereum.org/

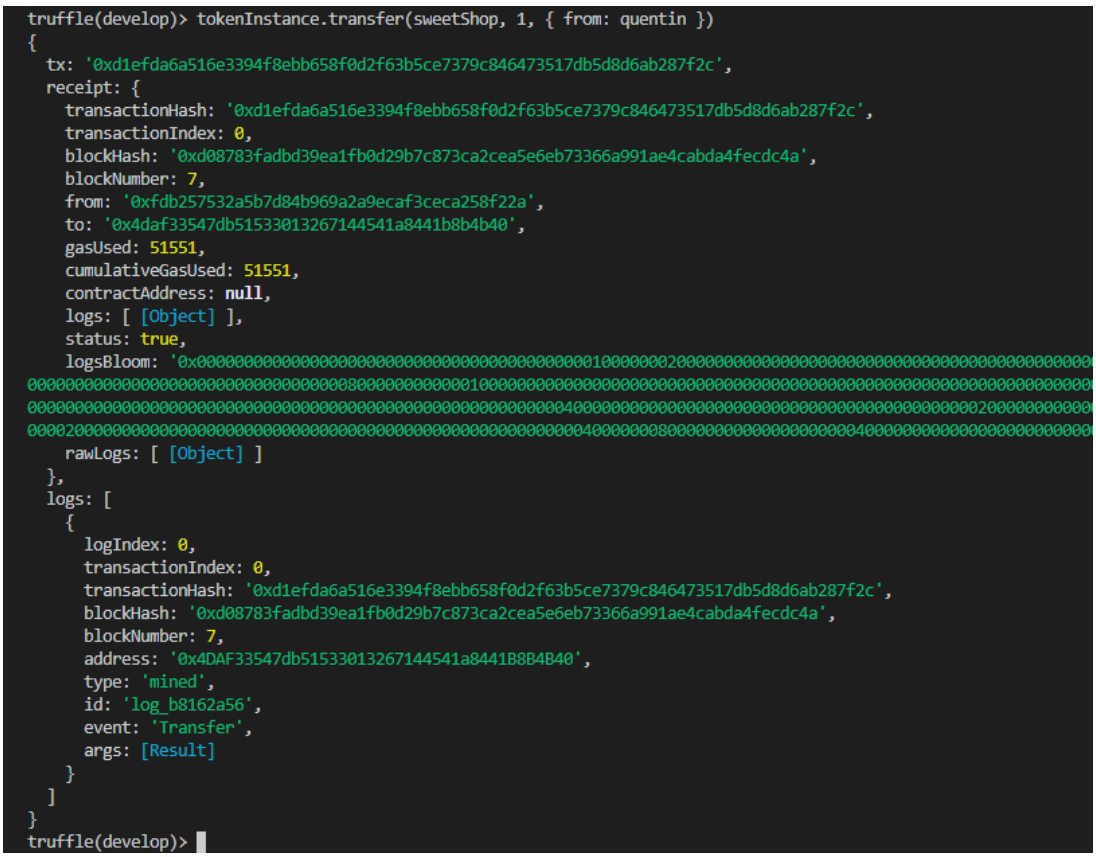

Figure 5: A transaction receipt is obtained upon successful transaction between two accounts!

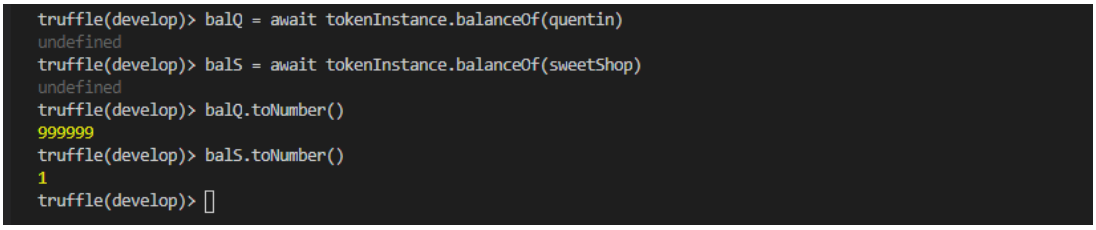

Figure 6: Account balances have changed due Quentin's purchase at the sweet shop.

4 LoanPlatformWithLoanTokens.sol

A smart contract was implemented with the same functions as outlined in section 2.1. A **setLoanTokens(loanTokenContract)** function was added, which is used to call the LoanToken contract and instanciate it within the current contract. Also, it makes use of the **Ownership** design pattern. This way, only the owner of the LoanPlatformWithLoanTokens contract can instantiate the LoanToken contract and call its methods.

All **payable** functions were modified to no longer remain payable. There is no need for a user to pass in an amount of Ether when using this smart contract, although gas fees still apply, as illustrated in figure 5. Also, the **transfer()** function was replaced by **token.transfer()** and **token.transferFrom()** where applicable.

5 Mocha/Chai Tests

Tests were written to ensure the LoanPlatform smart contract carries out its intended function as expected, and also that it reverts illegal transactions, giving the correct reason. The async-await style of testing was adopted. It is worth mentioning that each test references the same contract instance. Following is an exmaple of **desired** functionality being tested for:

The smart contract should transfer back the loan value and respective interest back to the lender and guarantor as soon as a borrower pays back the loan. The *LoanRequest* shown here has a *sum* of 2 Ether. The *guarantorInterest* and *lenderInterest* are both 0.25 Ether. Therefore, there must be a positive difference of 2.25 Ether in both the guarantor and lender accounts, as soon as the borrower has paid back.

```
221    it('should transfer back (funds + interest) to lender and guarantor upon payback', async () => {
222        instance = await LoanPlatform.deployed()
223        // get guarantor and lender balances before payback
224        let guarantorBalBefore = await web3.eth.getBalance(guarantor)
225        guarantorBalBefore =  web3.utils.fromWei(guarantorBalBefore, 'ether')
226        let lenderBalBefore = await web3.eth.getBalance(lender)
227        lenderBalBefore =  web3.utils.fromWei(lenderBalBefore, 'ether')
228        // payback loan
229        await instance.payBack(0, { from: borrower, value: web3.utils.toWei('2.5', 'ether')})
230        // get guarantor and lender balances after payback
231        let guarantorBalAfter = await web3.eth.getBalance(guarantor)
232        guarantorBalAfter =  web3.utils.fromWei(guarantorBalAfter, 'ether')
233        let lenderBalAfter = await web3.eth.getBalance(lender)
234        lenderBalAfter =  web3.utils.fromWei(lenderBalAfter, 'ether')
235
236        assert.equal((guarantorBalAfter - guarantorBalBefore).toFixed(2), 2.25)
237        assert.equal((lenderBalAfter - lenderBalBefore).toFixed(2), 2.25)
238    })
```

Figure 7: Testing for desired smart contract behaviour

Next is an example of **undesired** functionality being tested against. The smart contract should not allow guarantees to be stacked. In other words, once a guarantee is made, the borrower must accept or reject the guarantee and until then, no other guarantees can be placed on that particular *LoanRequest*. Try catch statements are used to check that undesired functionality is stopped by the smart contract. The reason why the transaction reverted is stored in *error.reason*, and asserted to be the same as the expected reason.

```
82     it('should block other guarantees once a guarantee has been placed but not yet accepted', async () => {
83         instance = await LoanPlatform.deployed()
84         try {
85             await instance.guarantee(0,
86                     web3.utils.toWei('0.25', 'ether'),
87                     { from: thirdParty, value: web3.utils.toWei('2', 'ether') })
88         } catch (error) {
89             assert.equal(error.reason, 'There is already a pending guarantee on this loan request!')
90         }
91     })
```

Figure 8: Testing for undesired smart contract behaviour

In total, 30 passing tests were written: 27 for LoanPlatform and 3 for LoanToken.

6 Reactjs User Interface with Web3

A simple user interface was developed for interaction with the LoanPlatform smart contract using Reactjs and Web3. Reactjs is a popular front-end library by Facebook, used for web and mobile application development. Web3 is a collection of libraries that enables interaction with a smart contract on a mainnet, public testnet (e.g., Rinkeby), or local testnet over HTTP, IPC or WebSocket.

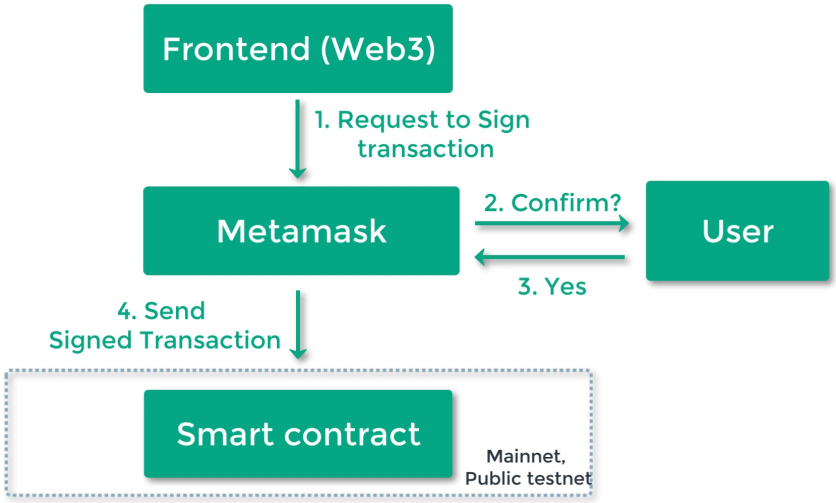

Figure 9: Frontend user interaction with the blockchain with Web3 and Metamask

To connect to a deployed smart contract and interact with it, Web3 needs two things:

- The contract Application Binary Interface (ABI), which describes the behaviour of the smart contract by listing all its functions, and their required arguments.
- The contract address.

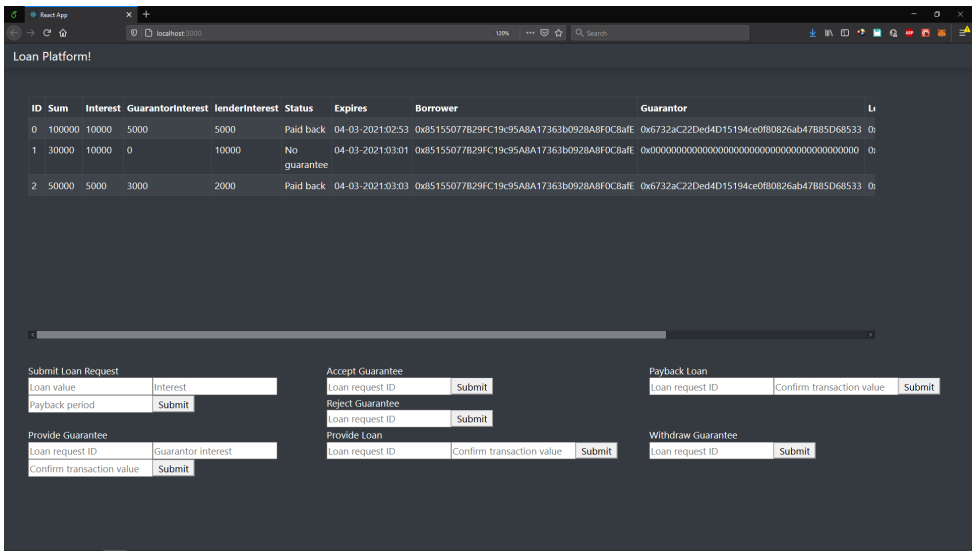

Figure 10: The UI

Three existing *LoanRequests* are fetched upon loading. This is because I have deployed the smart contract onto Rinkeby from Remix, and performed some transactions using my Metamask accounts and injected Web3. The top half of the UI is allocated for displaying all *LoanRequests*, and the bottom half is used for smart contract interaction. Each form field is validated before submission - Only numeric input is allowed.

- The *Submit Loan Request* form submission triggers a Metamask confirmation popup.

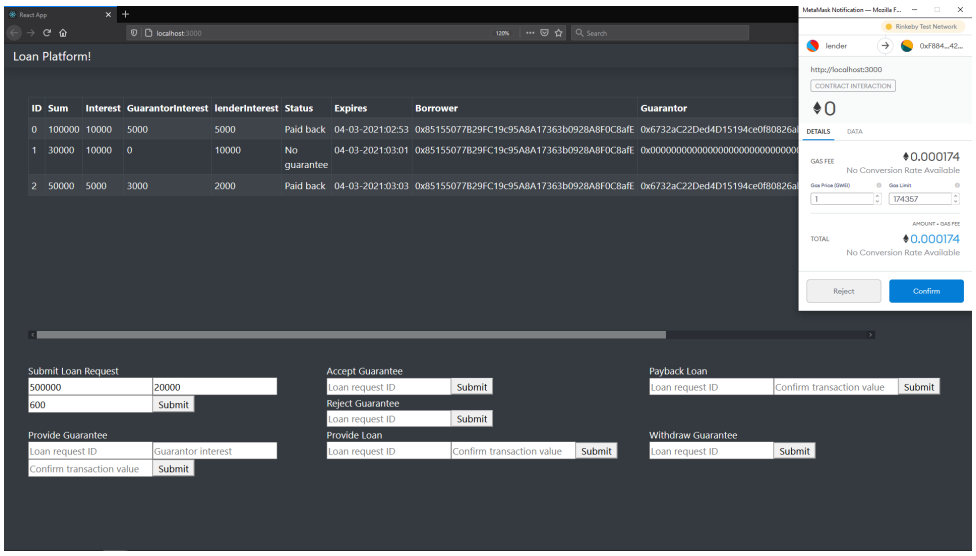

Figure 11: Submitting a loan request

- Upon refreshing, the newly submitted loan request appears.

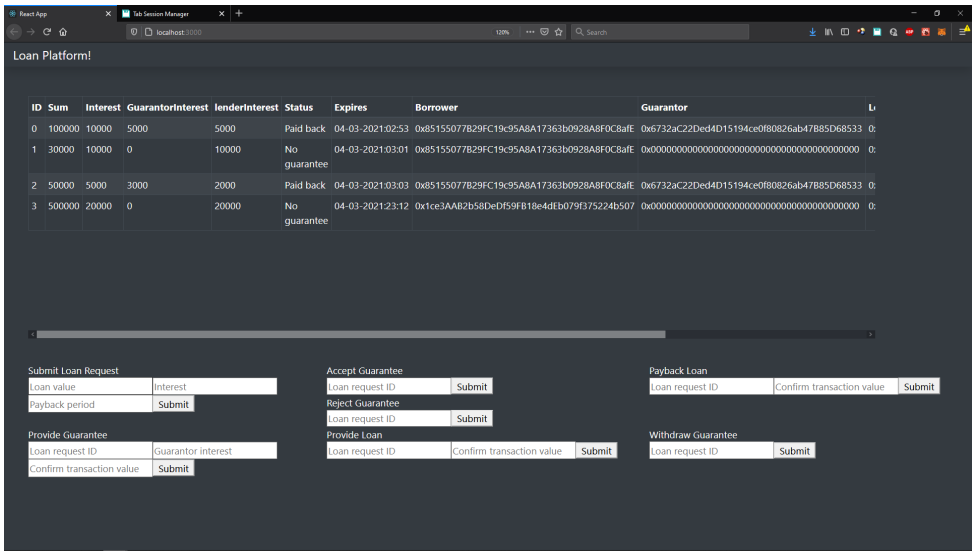

Figure 12: New request rendered on-screen

- From a different account, I provide a guarantee by filling submitting the *Provide Guarantee* form. Again, this causes Metamask to popup and ask for my confirmation.

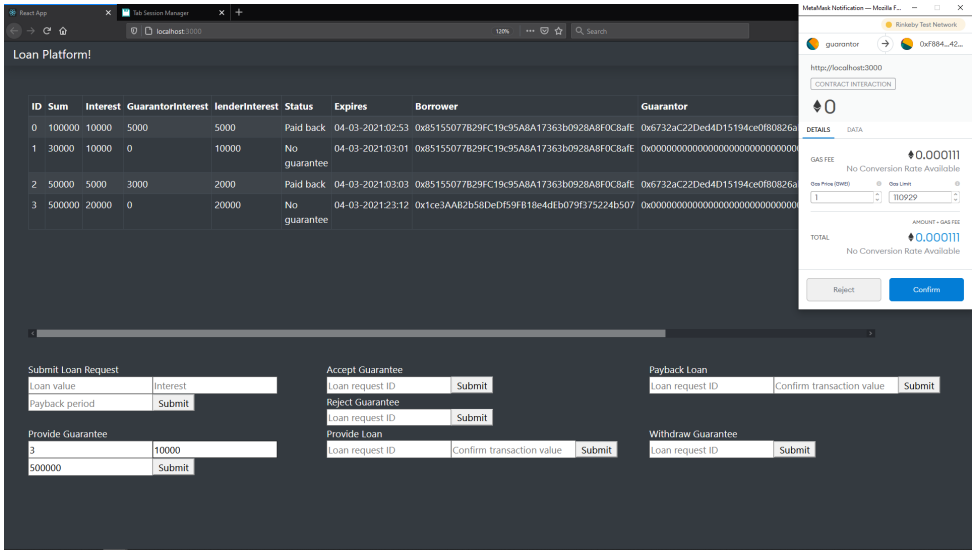

Figure 13: Guaranteeing a loan request

- Now the guarantor's address is associated with loan request ID 3, however the request is not marked as *guaranteed*. This will happen once the borrower *accepts* the guarantee.

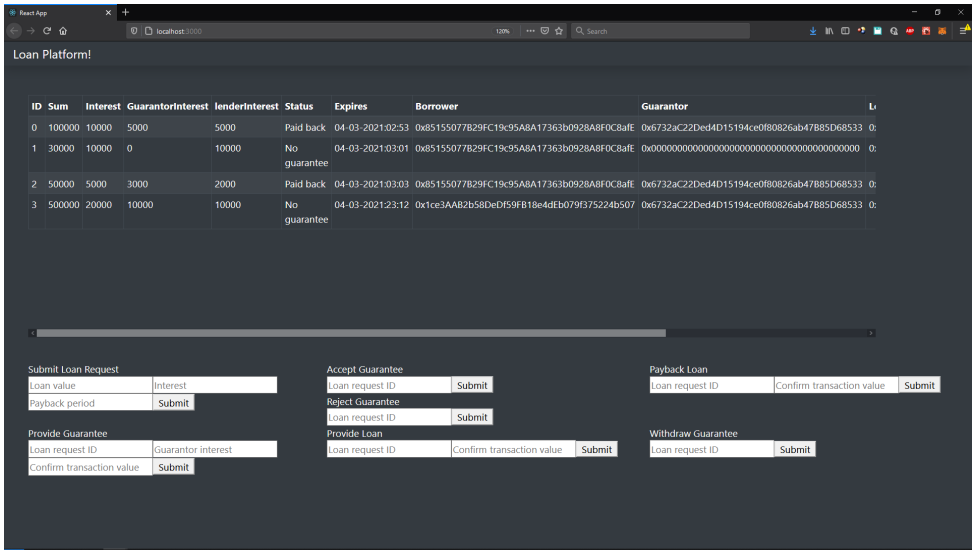

Figure 14: Guarantee received but not accepted by the borrower

I have demonstrated the functionality of the UI. Please find instructions on how to set up and use the UI further in the project README file.

7 Conclusion

Several design patterns were adhered to while developing the smart contracts, as outlined in sections 2.2 and 4. Reentrancy bugs were prevented in function logic by first performing relevant checks, then manipulating internal state and calling internal functions, and finally making external calls. Some points to improve on may include:

- The payback period starts counting down as soon as the borrower submits a request. In practice, it would make more sense for the payback time to start counting down once a loan has been provided by the lender.
- Loan requests which have been provided a loan, paid back or are otherwise expired should be hidden from the UI, as these are no longer relevant to neither of the three parties.