L-Università ta' Malta
Faculty of Information & Communication Technology

Assignment 1 Report

Manwel Bugeja

March 4, 2021

Contents

1 Problem 1

2 Flow of the Contract

The flow of the process is shown via a flowchart in figures 1 and 2.

2.1 Structures

The most fundamental part of the contract is a structure for loan requests, *LoanRequest*. This can be seen in listing 1. This structure contains the information required regarding the three persons included in the deal, i.e. the borrower, the guarantor and the loaner.

Listing 1: Loan Request Structure

```
struct LoanRequest{
        uint    creationDate;

        address payable borrower;
        uint    amount;             // Amount to be
           borrowed
        uint    expiryTime;         // Max time (in
           seconds) the loan will be paid back
        uint    interestPaid;       // Amount (in
           wei) paid on loan payback

        address payable guarantor;
        uint    guarantorInterest;  // Amount (in
           wei) taken from interest by the guarantor

        address payable loaner;
        State   state;
}
```

The last field of the loan request structure is an enumeration called *State*, which can be seen in listing 2. This enumeration contains the information about what state the loan request resides in. The states are explained in table 2.1.

Listing 2: Loan Request Structure

```
enum State {
        REQUESTED,
```

```
        PENDING,
        GUARANTEED,
        LOANED,
        PAID,
        TERMINATED
}
```

State	Description
REQUESTED	The borrower submitted a request
PENDING	The guarantor submitted a guarantee and is waiting for approval
GUARANTEED	The guarantee has been accepted by the borrower
LOANED	The loaner has loaned the money
PAID	The borrower paid the loaner back before the loan expired
TERMINATED	The loan expired and the loaner took the guarantee

These loan requests are all kept in a dynamic array, *LoanRequests[]*. Elements are added to this array from the functions. However, elements are never removed. This allows the system to keep a log of all requests ever made.

2.2 Functions

Functions in the contract are split into four sections, these being: general purpose, borrower, guarantor and loaner functions.
General purpose functions are mostly used to get information about the loan requests. This means that most of them include the *view* keyword.
Functions that act on a created loan request take the index as a parameter. Furthermore, the functions have the following flow: checks, followed by code that transfer balance, followed by updating the loan request states. The reason for this is explain in the section about security.

2.3 Testing and Error checking

The functions that are to be executed by users of a specific role (borrower, guarantor or loaner) have checks at the start to ensure validity. These checks are done using the *require* function so useful messages are outputted.
For example, referring to listing 3 when submitting a guarantee the check done are: checking that the request is in state *REQUESTED*; checking that the amount of balance given by the guarantor is more that the amount requested by the borrower; and finally, checking that the interest to be taken by the guarantor is less than the interest the borrower will provide.

Listing 3: Loan Request Structure

```
require(loanRequests[index].state == State.REQUESTED
   ,   "Loan␣in␣invalid␣state");
require(msg.value >= loanRequests[index].amount,
          "Insufficient␣balance␣given");
require(loanRequests[index].interestPaid > interest,
       "Guarantor␣interest␣must␣be␣less␣than␣
   Borrower␣interest");
```

Testing the functions in task 1 was done manually via the Remix IDE. The functionality was tested to be working as should with the provided interface and that in the case of misuse, a useful error message was printed. The feature to use multiple account was also used.
For example if a guarantor wanted to try and accept his own guarantee, the checks would not pass and the error message "Caller is not Borrower" would provided as shown in figure 3.

3 Problem 2

4 Problem 3

JavaScript tests were written using the *async* method. Tests were conducted for all functions and different accounts were used. Listing 4 show how multiple accounts were used to simulate multiple people.

Listing 4: Multiple accounts in testing

```
contract('MyContract', accounts => {
        const borrower = accounts[0];
        const guarantor = accounts[1];
        const loaner = accounts[2];
        ...
}
```

Tests that were made manually om the remix IDE were replicated with these types of tests. Considering the same example as before, where a guarantor might try to accept his own guarantee, the test is shown in listing 5.

Listing 5: Guarantor accepting his own guarantee test

```
it('testing␣a␣guarantor␣accepting␣his␣own␣guarantee'
   , async() => {
        const contract = await MyContract.deployed()
           ;
        await contract.submitLoanRequest(200,
           100000, 20, {from: borrower});
        await contract.guaranteeLoan(3, 10, {value:
           500, from: guarantor});

        try {
                await contract.acceptGuarantee(3, {
                   from: guarantor});
        } catch (error) {
                assert.equal(error.reason, "Caller␣
                   is␣not␣Borrower")
        }
});
```

Listing 6: Borrower submitting a load request

```
it('testing␣submitting␣a␣loan␣request', async () =>
   {
        const contract = await MyContract.deployed()
           ;
        await contract.submitLoanRequest(200,
           1614786990, 20, { value: 0, account:
           borrower });
        await contract.submitLoanRequest(300,
           1614786990, 18, { value: 0, account:
           borrower });
        let ret = await contract.getLoanRequestCount
           .call();
        assert(ret.toNumber() == 2);
});
```

Tests were conducted for all functions that change the state of the contract i.e. all of them except the getters. There are several tests for each function because there are several invalid ways to use a function and only one good/valid way. The tests test for all invalid uses hence triggering all *require* calls. This enables the system to be more robust.

5 Problem 4

First off, the file *truffle-config.js* was edited to add metamask and infura. The rinkeby live test network was used. Then npx was installed and a react app was created.

Then *App.js* was edited to display the required fields for contract interaction. The abi was copied from *build/contracts/myContract.json*.
Listing 7 show the back end of submitting a loan request. While figures 4 and 5 show submitting and confirming a loan request from the web ui. After that, when the transaction is successfully updated, the count above the form is also updated.

Listing 7: Submitteing a borrow request from web

```
    submitLoanRequest = async (event) => {
          event.preventDefault();
          var accounts = await web3.eth.
             getAccounts();
          var myContract = myContract = new
             web3.eth.Contract(abi,
             contractAddress);
          await myContract.methods.
             submitLoanRequest(this.state.
             loanRequestAmount, this.state.
             loanRequestTime, this.state.
             loanRequestInterest).send({
                from: accounts[this.state.
                   accountNum]
          });
          this.updateState();
    }
```

Due to bad time management from my end, only two of the contract functions were implemented but they can manually be tested to be working as should.
Having managed my time better, I would have proceeded to implement the other functions in a very similar manner. After that I would then have continued by making the errors resulting from *require* to be shown on the page. This would have been done using the *try; catch* keywords similar to the chai tests.

6 Problem 5

For this problem please refer to figure 6.

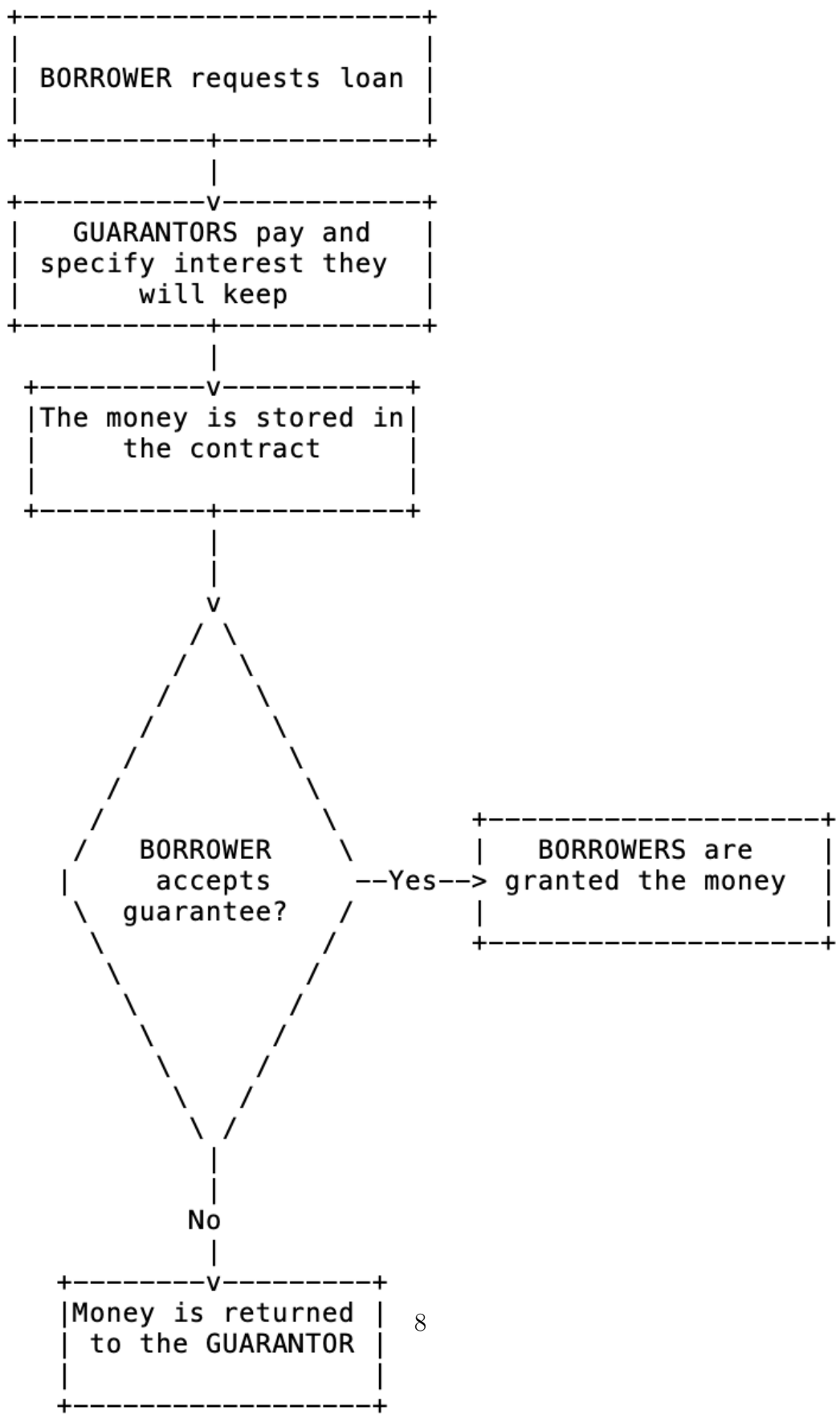

Figure 1: Submitting a Loan Request

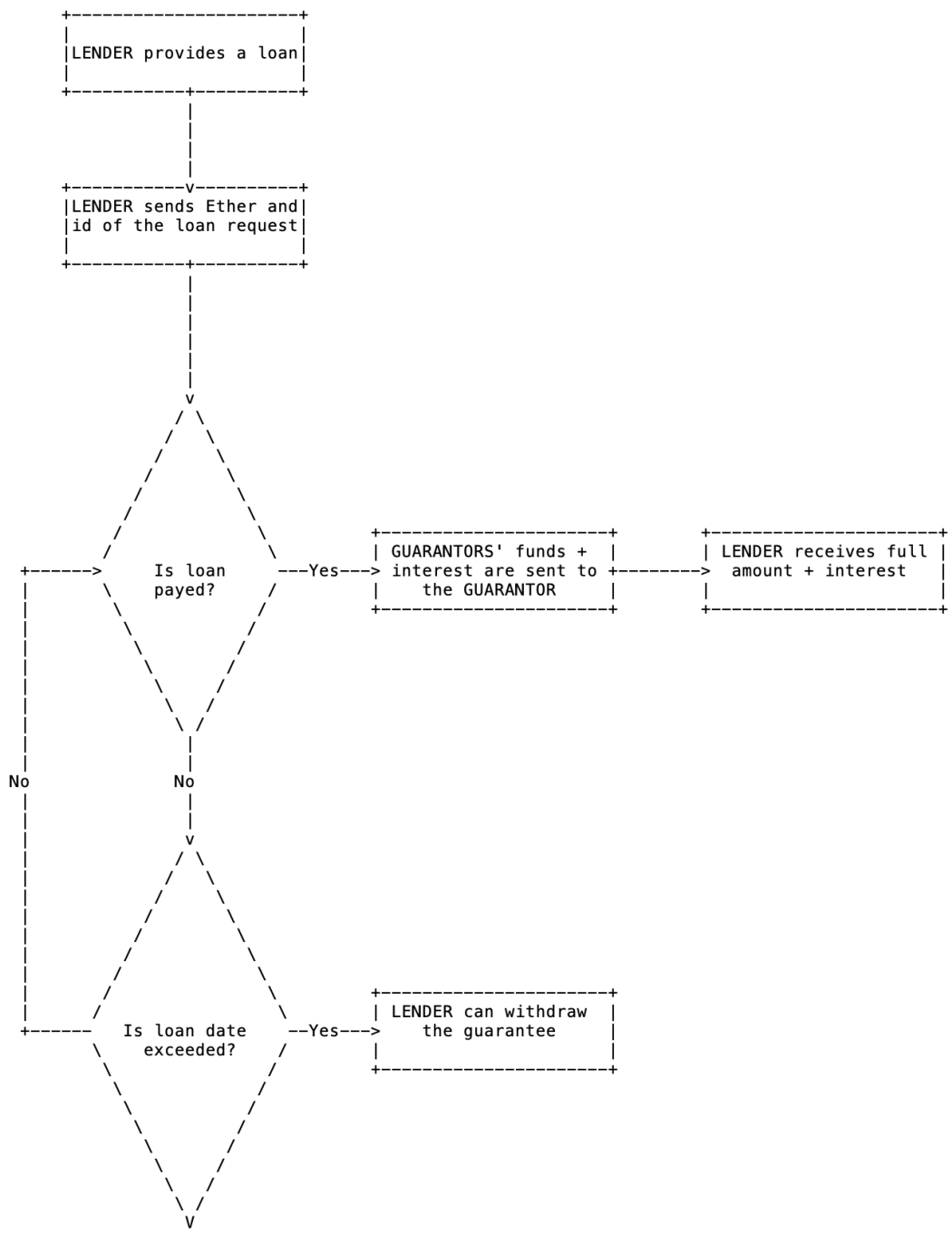

Figure 2: Submitting a Loan

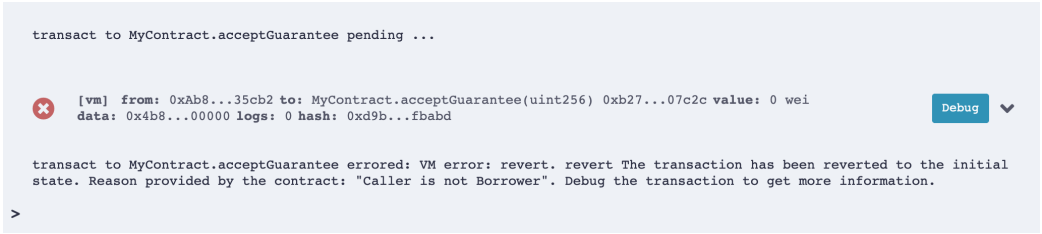

Figure 3: Guarantor Accepting his Own Guarantee

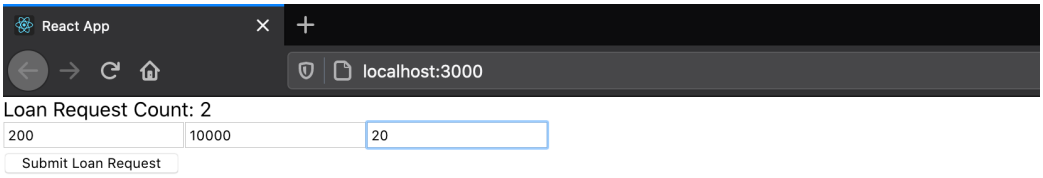

Figure 4: Because question 5 is the report itself

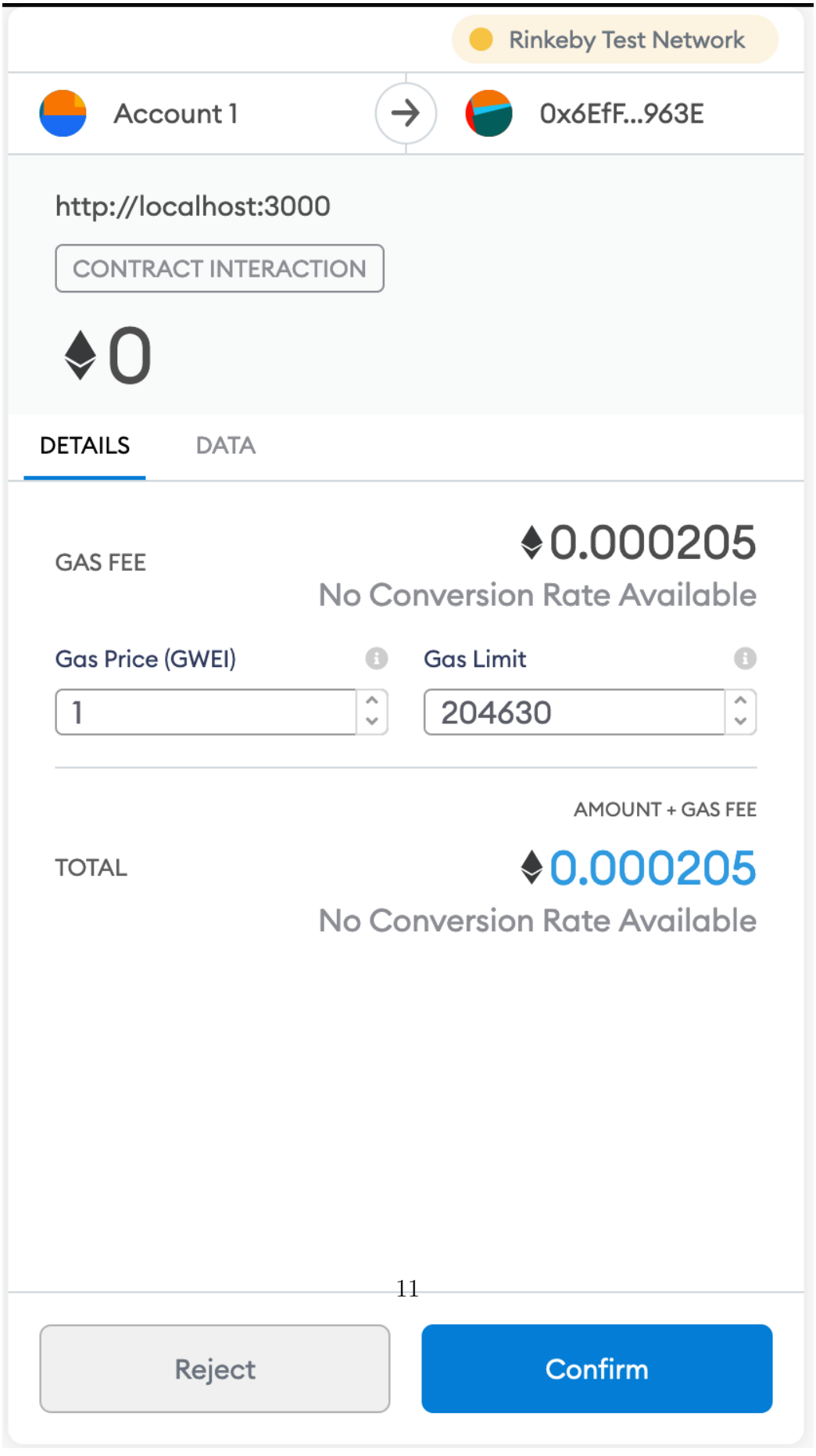

Figure 5: Because question 5 is the report itself

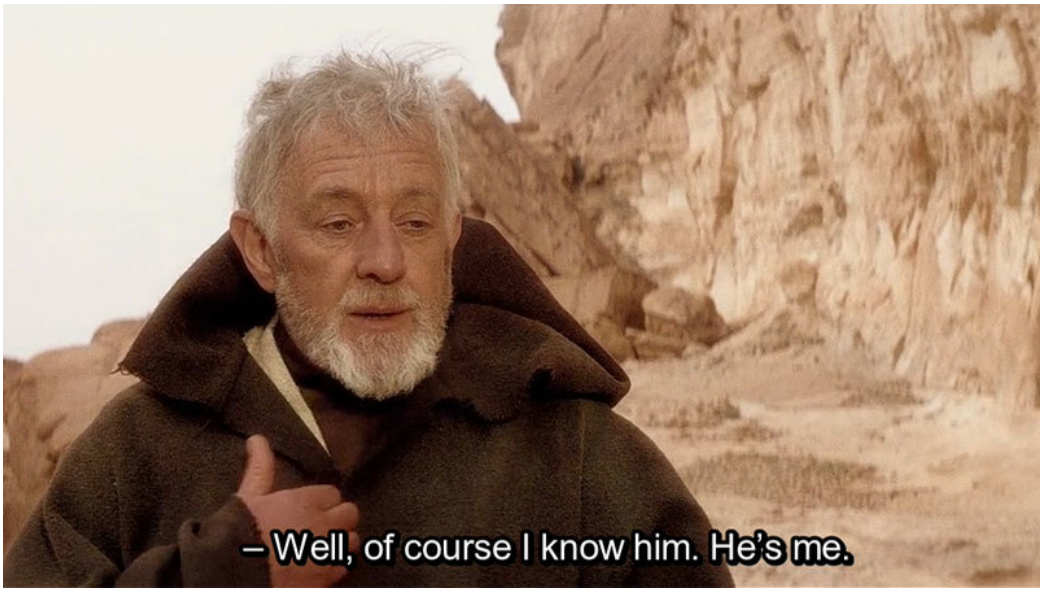

Figure 6: Because question 5 is the report itself