**CarContract readme**

**Author: S.A. Stokes**

**Date: 2018.02.11**

This document describes the intended functionality of a smart contract called CarContract. The contract is written in Solidity and meant to run on Quorum, a permissioned blockchain based on ethereum.

CarContract is meant to keep a history of all maintenance performed on a car, beginning with the car’s inception at the manufacturer. A car owner maybe interested in this information if they want to look at the average number of miles a particular brand of tire usually lasts on their vehicle, when was the last time they replaced their timing belt, or if they want to identify the mechanic who faultily installed a particular part. By storing this information on the public Ethereum blockchain, the authenticity of the information is ensured. Before the car is sold, the potential buyer can view the entire car history and feel more comfortable assigning an accurate value to the car. Please view the “Areas for Improvement” section at the end of this document for a description of the next features that need to be added to CarContract.

Migration file: 4\_deploy\_carcontract.js

privateFor

Currently the CarContract is deployed with a privateFor parameter containing the address of Node2’s Constellation public key. This means that only Node1 (the initiator of the contract) and Node2 (a recipient of the contract) can see the unencrypted data of the contract. The encrypted data gets stored on the Ethereum blockchain, but no one outside of Node1 and Node2 can discern what the data means. An important note: privateFor must be passed with every transaction. The privateFor argument(s) in the transactions must be the same as what is used in the migration file when first creating the contract, otherwise the transaction looks like it is trying to execute against a completely different contract.

Smart contract file: CarContract.js

These are the functions contained in CarContract and the expected parameters (inputs) and return values (outputs) for each.

**function CarContract**: constructor. Initializes a vehicle’s static information and sets the first owner of the car to whoever called the contract.

Inputs:

* uint64 vin\_: Vehicle Identification Number (VIN). This is a 17 digit number and every vehicle has its own unique number.
* string newCarMake: The name of the car manufacturer, i.e. Honda.
* string newCarModel: The type of car, i.e. Accord 1998.

Outputs: N/A

**function getAllIndexes**: Retrieves the array positions for the most recent owner and installed parts.

Inputs:

* uint64 vin\_: Vehicle Identification Number (VIN).

Outputs:

* uint8 ownerIndex
* uint8 airbagIndex
* uint8 engineIndex
* uint8 tiresIndex

**function getStaticCarInfo**: Retrieves the vehicle information that does not change after the constructor is called.

Inputs:

* uint64 vin\_: Vehicle Identification Number (VIN).

Outputs:

* uint64 vinOut: Vehicle Identification Number (VIN)
* string \_carMake: The name of the car manufacturer, i.e. Honda.
* string \_carModel: The type of car, i.e. Accord 1998.

**function get<partName>**: Part name can be “Airbag”, “Engine”, or “Tires”. Gets all of the stored part information at the array position specified by an index parameter.

Inputs:

* uint64 vin\_: Vehicle Identification Number (VIN).
* uint8 arrayIndex\_: Position in array to retrieve part information from

Outputs:

* string \_part: the part number
* uint \_timeInstalled: the time when the part was installed (seconds since 1970.01.01)
* uint \_carMileage: how many miles the car had when the part was installed
* string \_whoInstalledPart: name of the mechanic and shop where the part was installed

**function change<partName>**: Part name can be “Airbag”, “Engine”, or “Tires”. Appends a new part to the end of a part array. Previous parts’ information is preserved in the array. The owner of the vehicle is the only one who can make a part change so this function checks that the sender of the request matches the owner address attached to the vehicle.

Inputs:

* uint64 vin\_: Vehicle Identification Number (VIN).
* string newPart: Part number
* uint currentMileage: Mileage of vehicle when part is changed
* string newMechanic: Name of the mechanic who changed the part

Outputs: N/A

**function changeOwner:** This functions changes the owner address attached to the given VIN. Only the current owner can change the owner address. After it has been changed, the previous owner will lose the ability to edit any of the vehicle information. The new owner becomes the one with sole access to make transactions against the contract.

Inputs:

* uint64 vin\_: Vehicle Identification Number (VIN).
* address newOwner: Ethereum public key of the new owner
* uint currentMileage: Mileage of the vehicle when the ownership changed

Outputs: N/A

**function getOwner:** Works the same as get<Part> except it returns owner information instead of part information.

Inputs:

* uint64 vin\_: Vehicle Identification Number (VIN).
* uint8 arrayIndex\_: Position in array to retrieve owner information from

Outputs:

* string \_part: the part number
* uint \_timeInstalled: the time when the part was installed (seconds since 1970.01.01)
* uint \_carMileage: how many miles the car had when the part was installed
* string \_whoInstalledPart: name of the mechanic and shop where the part was installed

Test file: testCarContract.js

The following list shows the tests run against CarContract.sol, using asynchronouse Node JS code with promises. Put testCarContract.js in the truffleProject/test/ directory and run the following command from the truffle project root directory to execute the tests: truffle test ./test/testCarContract.js

1. Retrieve the static vehicle information and verify that it matches the information sent to the constructor.
2. Verify that the initial owner’s address matches the Ethereum public key of the account that created the contract.
3. Add an airbag part to the vehicle and retrieve that information using getAirbag().
4. Attempt to add a part to the vehicle from an authorized account. Verify that the part is not added to the car’s log.
5. Create two different installations of tires to the vehicle and then read back the full tire history.
6. Change the owner of the vehicle and verify that the old owner can no longer add parts to the car’s log.

Areas for improvement:

CarContract is incomplete because it only holds information about three different parts. However, the functions to get and change parts could easily be replicated for all parts in the vehicle. Similarly, functions could be created to get and set information pertaining to emissions certification, inspections, and accidents. Perhaps there could be an insurance structure added for each vehicle and the smart contract could be private between the car owner and the insurance company.

When changing ownership, I would like to deploy a new contract so that the privateFor parameter can be changed. I do not believe that the privateFor public keys list can be changed in any way after the contract is deployed, therefore a completely new contract would be necessary. I think one way to accomplish this would be to call a new contract from within CarContract whenever the changeOwner() function is called. All of the information from the previous contract could be copied to the new one. This would keep the vehicle information private so that only the current owner can see the vehicle state changes while they are the owner. When they sell the vehicle, they will lose the ability to view future state changes for that vehicle. I could even kill the old contract using the suicide() function to clear all of the old contract’s data from the blockchain.

“testCarContract.js” is currently unable to send transactions from any account except accounts[0] from Node1 of the test network. Attempted transactions from accounts[1]—accounts[3] produce the following error: “authentication needed: password or unlock”. In order to test this functionality, the other accounts would have to be unlocked using their Ethereum private keys. I believe this can be accomplished using “personal.unlockAccounts(accountPubAddress, accountPassword, 0);” but I have not yet figured out where or how to call this function.

“testCarContract.js” is mostly hard-coded to test specific elements of the contract. I haven’t figured out how to use loops within promises. This would be helpful when looking at a particular part’s history. I would like to run the getIndexes() function to determine how many instances of that part have ever been installed on the vehicle. Using that number, I could run a loop to getPartInfo starting at index 0 and running until I have retrieved information for every instance of the part type.