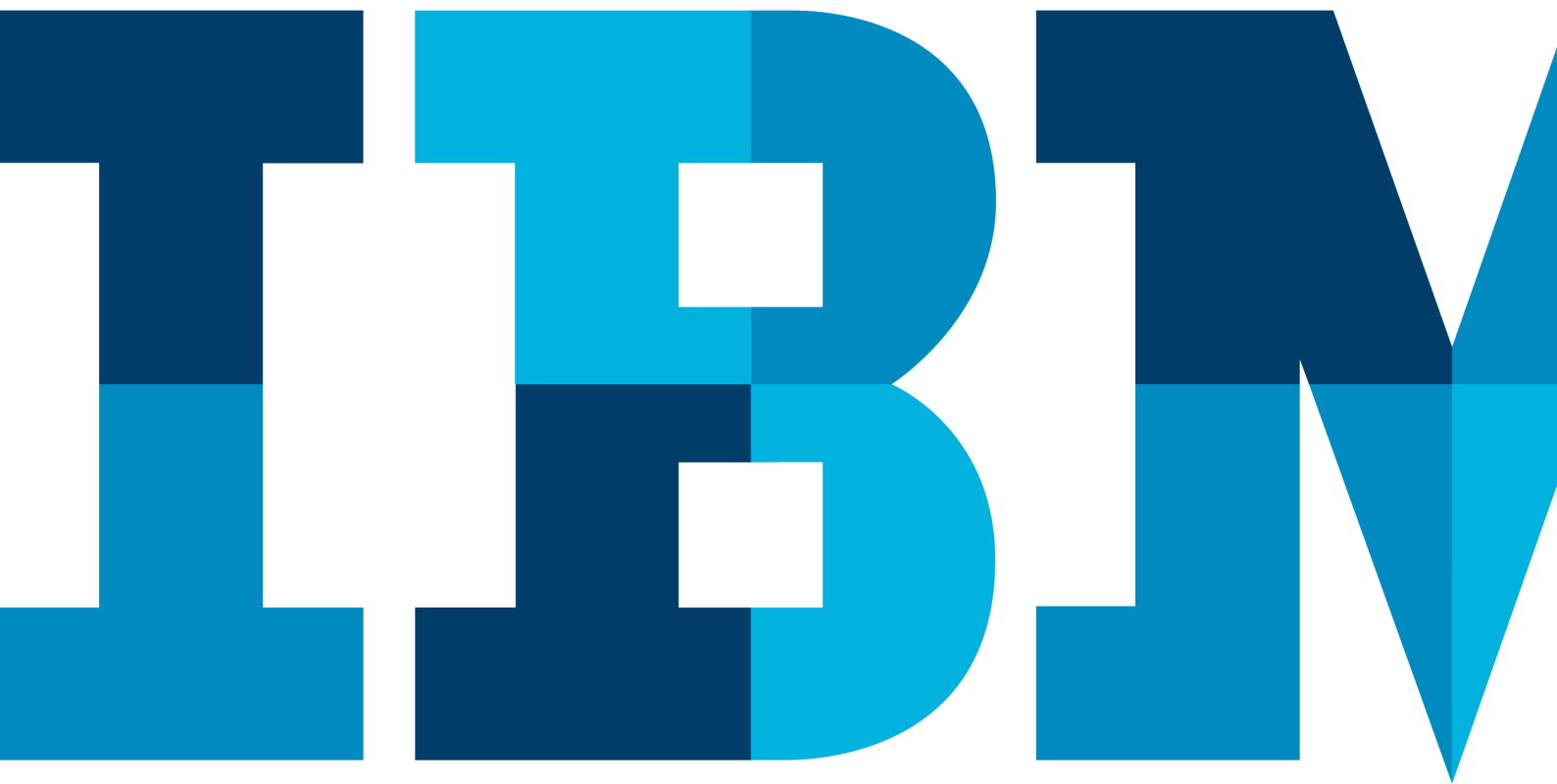


# IBM Blockchain Proof of Technology Demo Assets Transfer Demo - Deep Dive

*Lab Three – VM - Exercises*



# Contents

<b>SECTION 1. INTRODUCTION .....</b>	<b>3</b>
1.1 TIERED ARCHITECTURE.....	4
1.1.1 <i>User Interface</i> .....	5
1.1.2 <i>API Layer</i> .....	5
1.1.3 <i>Blockchain Runtime</i> .....	6
<b>SECTION 2. SCENARIOS .....</b>	<b>7</b>
2.1 SCENARIO: TRANSFER: MANUFACTURER TO DEALERSHIP .....	7
2.1.1 <i>Select Vehicle to Transfer</i> .....	7
2.1.1.1 User Interface Tier.....	8
2.1.1.2 API Tier.....	8
2.1.1.3 Blockchain Tier .....	10
2.1.2 <i>Select Dealership</i> .....	11
2.1.3 <i>Transfer Asset</i> .....	12
2.2 SCENARIO: CREATE VEHICLE TEMPLATE (REGULATOR).....	18
2.3 SCENARIO: VIEW BLOCKCHAIN ACTIVITY.....	20
<b>SECTION 3. CHAINCODE IN BLOCKCHAIN .....</b>	<b>23</b>
3.1 CHAINCODE OVERVIEW .....	23
3.2 VEHICLE CHAINCODE .....	24
3.3 VEHICLE FUNCTIONS .....	25
3.3.1 <i>Constructor</i> .....	25
3.3.2 <i>Validation functions</i> .....	25
3.3.2.1 GET_ECERT.....	25
3.3.2.2 CHECK_ROLE .....	26
3.3.3 <i>Invoke functions</i> .....	27
3.3.3.1 CREATE_VEHICLE .....	27
3.3.3.2 AUTHORITY_TO_MANUFACTURER .....	28
3.3.3.3 UPDATE_VIN .....	29
3.3.4 <i>Query functions</i> .....	31
3.3.4.1 GET_ALL.....	31
3.3.5 <i>Chaincode invoking chaincode</i> .....	31
3.4 VEHICLE_LOG CHAINCODE.....	32
3.5 VEHICLE_LOG FUNCTIONS .....	33
3.5.1 <i>Constructor</i> .....	33
3.5.2 <i>Validation Functions</i> .....	33
3.5.3 <i>Invoke Functions</i> .....	33
3.5.3.1 CREATE_LOG.....	33
3.5.4 <i>Query functions</i> .....	34
3.5.4.1 GET_LOGS .....	34
3.5.4.2 GET_USERS_LOGS .....	35

## Section 1. Introduction

This guide provides a description of the technical aspects of the IBM Blockchain Proof of Technology (PoT) demo and should be used in conjunction with the “IBM Blockchain Proof of Technology, Demo Scenarios” Lab guide.

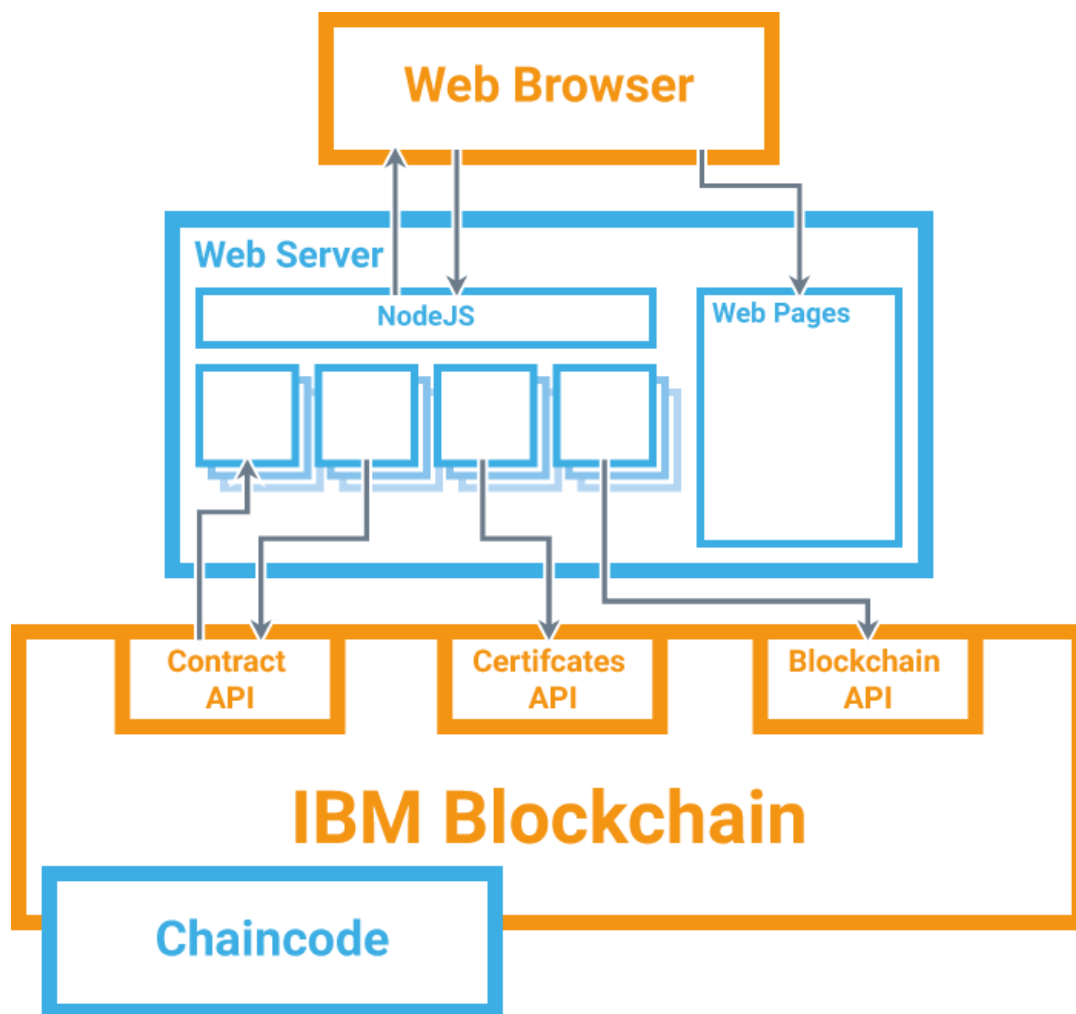
The purpose of this guide is to present an explanation of how certain functions are implemented in the demo. Source code presented in this document has been truncated where appropriate to ensure brevity. As a result, the source code contained in this document should be considered incomplete: for a full listing, refer to the PoT VMware demo environment. Where the discussion in this guide is limited due to space constraints, the reader is invited to further explore the code base and experiment with the Blockchain functionality.

## 1.1 Tiered Architecture

The PoT demo implements a three-tiered architecture. The tiers are:

1. User Interface;
2. Blockchain Demo API;
3. IBM Blockchain Runtime.

The figure below depicts the component architecture.



The User Interface is implemented as a Web client consisting of application code written in HTML and Javascript and is primarily intended to provide a view of, and means of managing, participant accounts and contracts throughout the asset management lifecycle. The Web client interacts with the Blockchain Demo API layer by making REST calls to a Node.js server. This Node.js server implements the API layer and maps each method to a set of calls that are then sent to the *IBM Blockchain* runtime. This Blockchain runtime provides the core functionality of the Blockchain and manages the state of the Blockchain throughout the asset management lifecycle. Programmatic elements (that represent vehicles and contracts in the supply chain scenario) are encoded in *Chaincodes* and are invoked via the Blockchain runtime.

## 1.1.1 User Interface

The Web pages listed below implement the user interface for the following core functions:

### Control screens

`index.html`

`admin-console.html`

`stats.html`

### Edit Vehicle

`manufacturer-update.html`

### Asset Transfer

`manufacturer.html`

`dealership.html`

`lease-company.html`

### Leasing

`leasee.html`

### Scrapping the Vehicle

`scrap-merchant.html`

### Audit

`regulator-view.html`

These pages are located in **/home/ibm/Documents/Demo/Client\_Side**. Each page depends on one or more of the following Javascript files:

`config.js`

`identity.js`

`general_page_functions.js`

`update_page_functions.js`

`charts.js`

`stats.js`

`asset_interaction.js`

`asset_read.js`

`scrollbar.js`

`recipients.js`

`vehicles_popup_functions.js`

`recipient_popup_functions.js`

`page_functions.js`

`ledger.js`

Additional pages are located in the same directory and are used to display information during the appropriate stage in the asset transfer lifecycle. For instance, **manufacturer.html** presents a screen during the transfer of a vehicle from manufacturer to dealership.

These pages also depend on various third-party libraries, including jQuery, JQuery-UI and D3.

The Javascript files are located in **/home/ibm/Documents/Demo/Client\_Side/JavaScript** or in subdirectories thereof.

## 1.1.2 API Layer

The API layer is implemented by files located in **/home/ibm/Documents/Demo/Server\_Side**. The API layer consists of a router component that receives and forwards requests to the Blockchain tier.

The majority of the files that implement the router are located in **/home/ibm/Documents/Demo/Server\_Side/blockchain**, with some additional files located in **/home/ibm/Documents/Demo/Server\_Side/admin** and **/home/ibm/Documents/Demo/Server\_Side/tools**. An additional file, *router.js*, acts as a service façade for receiving requests from the user interface tier.

These files are categorized according to their core role: for instance, functionality for retrieving and formatting information on participants in the Blockchain are stored under the **blockchain/participants** sub-folder.

### 1.1.3 Blockchain Runtime

The IBM Blockchain runtime performs the core operational tasks for managing the lifecycle of the Blockchain. Throughout this document, this operational capability is abstracted through a simple interface that involves contracts and accounts. Detailed introspection of the Blockchain runtime internals is outside the scope of this document.

Further details of the IBM Blockchain project and access to source code may be found at

<https://github.com/openblockchain>

## Section 2. Scenarios

In this section, a detailed examination of a selection of scenarios from the Blockchain PoT demonstrator is provided.

### 2.1 Scenario: Transfer: Manufacturer to Dealership

To recap, this scenario transfers the ownership of a vehicle from a Manufacturer to a Dealership (known as “Beechvale Group”) using the Blockchain. Both the Manufacturer and the Dealership are selected at the time of transfer.

#### 2.1.1 Select Vehicle to Transfer

Interactions between components in the three tier architecture are summarized in Figure 1.

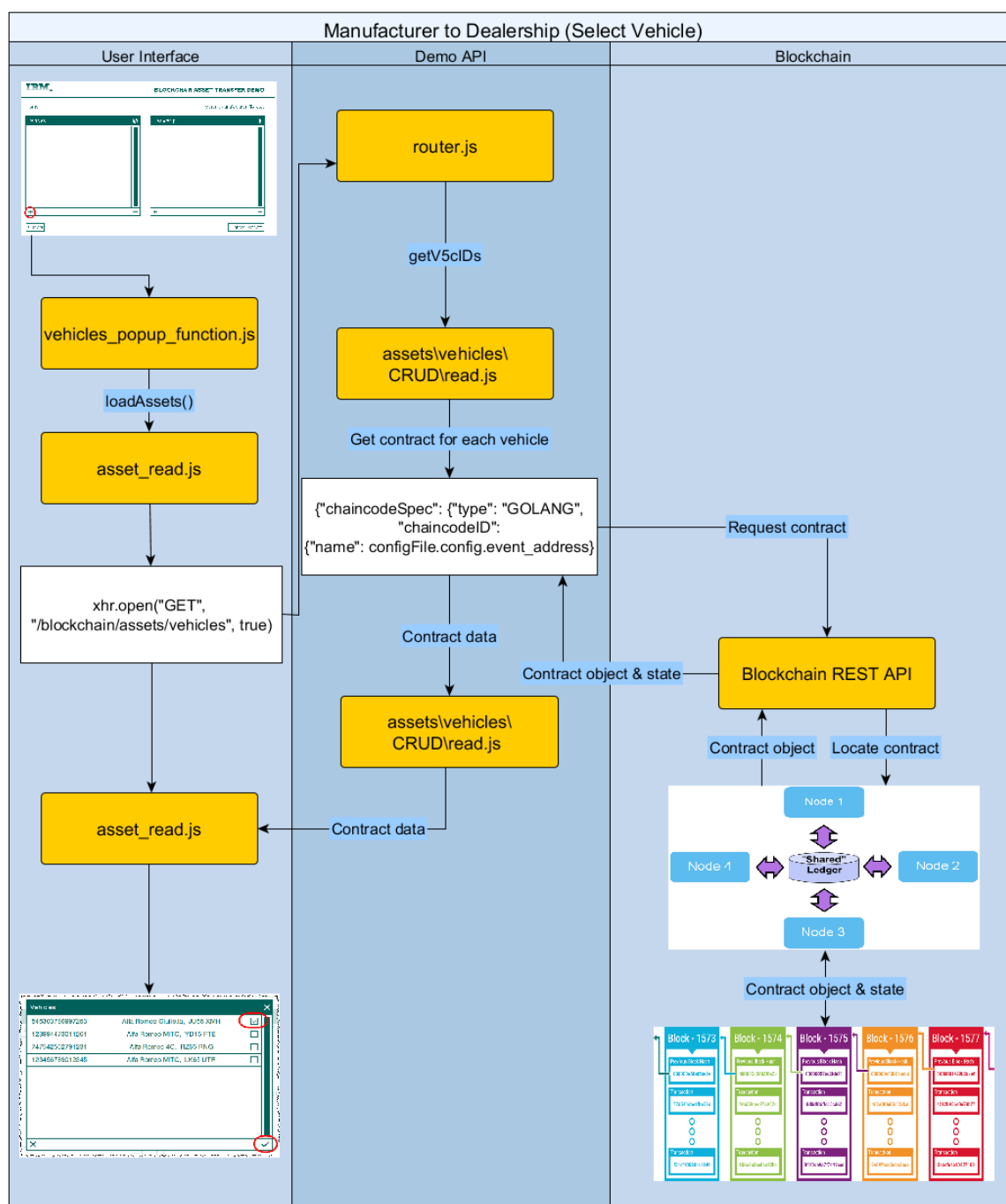


Figure 1 -- Asset Transfer Scenario (Select Vehicle)

### 2.1.1.1 User Interface Tier

Clicking the  icon defined in **manufacturer.html** calls the following function located in **vehicles\_popup\_functions.js**:

```
function() {
    setCookie();
    $('#vhclsTbl').html('<tr><td style="text-align:center;"><br
/><i>Found: <span class="numFound">0 assets</span></i></td></tr>');
    $('#chooseOptTbl').fadeIn(1000);
    $('#fade').fadeIn(1000);
    loadAssets();
}
```

This function prepares the display for a list of assets (vehicles) from which to transfer between manufacturer and dealership. The function then calls the **loadAssets** function in **asset\_read.js** which sends a HTTP GET request to the API tier (of the 3-tier architecture) and specifically to **/blockchain/assets/vehicles**.

```
xhr.open("GET", "/blockchain/assets/vehicles", true)
```

From the response, all vehicles that are currently allocated to the manufacturer are extracted:

```
if(objects[j].v5cID == obj.v5cID)
{
    found = true;
    break;
}
```

The final list of vehicles is displayed on page and is ready for the user to make a selection.

### 2.1.1.2 API Tier

The API tier receives the request from the Web client and is processed by the following function defined in **router.js**:

```
app.get('/blockchain/assets/vehicles/' , function(req,res)
{
    vehicles.read(req,res)
});
```

This calls the **read** method implemented in **assets/vehicles/CRUD/read.js** -- note that the method's interface is declared separately in **vehicles.js**. This method calls the following function:

```
getV5cIDs(req, res)
```

This function, in turn, invokes the Blockchain runtime tier by calling an http post request on the API at:

```
/devops/query
```

And passing in the body:

```
var chaincodeInvocationSpec = {
    "chaincodeSpec": {
        "type": "GOLANG",
        "chaincodeID": {
            "name": configFile.config.event_address
        },
        "ctorMsg": {
            "function": "get_vehicle_logs",
            "args": [req.session.user]
        },
    },
}
```



```

        "secureContext": req.session.user,
        "confidentialityLevel": "PUBLIC"
    }

```

The Blockchain runtime tier itself exposes a REST API which presents an interface consisting of verbs for managing the Blockchain lifecycle. Details of this REST API can be found in [github.com/openblockchain/obc-peer/openchain/rest](https://github.com/openblockchain/obc-peer/openchain/rest)<sup>1</sup> of the IBM Blockchain project.

Examples of common uses of the IBM Blockchain API include:

- retrieving information regarding the state of the Blockchain and of individual blocks in the Blockchain;
- searching for transactions and retrieve information relating to specific transaction instances;
- deploying chaincode and query chaincode state;
- invoking functions defined in chaincode;
- registering a user with the certificate authority and confirm that a user is registered;
- retrieving enrollment certificates;
- retrieving information about network peer connectivity.

The appropriate chaincode must be identified (by *chaincodeID*): the specific chaincode in this instance is identified by the value set by **configFile.config.event\_address**. The chaincode is queried with the following code:

```

var options = {
    url: configFile.config.api_url+'/devops/query',
    method: "POST",
    body: chaincodeInvocationSpec,
    json: true
}

request(options, function(error, response, body)
{
    if (!error && response.statusCode == 200)
    {
        for(var i = data.length-1; i > -1; i--)
        {
            if(data[i].name == "Create")
            {
                var v5cID = data[i].obj_id;
                ids.push(v5cID);
            }
        }
        if(ids.length > 0)
        {
            getV5cDetails(ids, 0, req, res)
        }
        ...
    }
}

```

This code examines each *vehicle\_log* returned from the Blockchain and for those that represent vehicles being created it will request further details of that vehicle (by invoking the **getV5cDetails** function). In turn, the **getV5cDetails** function invokes a service on the Blockchain API using:

```
/devops/query
```

<sup>1</sup> Source code for the IBM Blockchain project can be located on GitHub.

Passing in the body:

```
var chaincodeInvocationSpec = {
    "chaincodeSpec": {
        "type": "GOLANG",
        "chaincodeID": {
            "name": configFile.config.vehicle_address
        },
        "ctorMsg": {
            "function": "get_all",
            "args": [req.session.user,
                    ids[i].toString()]
        },
        "secureContext": req.session.user,
        "confidentialityLevel": "PUBLIC"
    }
}
```

... and then the V5C certificate for each vehicle is examined:

```
var options = {
    url: configFile.config.api_url+'/devops/query',
    method: "POST",
    body: chaincodeInvocationSpec,
    json: true
}

request(options, function(error, response, body)
{
    if (!error && response.statusCode == 200)
    {
        var resp = body.OK;
        resp.v5cID = ids[i];
        res.write(JSON.stringify(resp)+'&&');
        if(i < ids.length -1)
        {
            getV5cDetails(ids, i+1, req, res);
        }
        ...
    }
}
```

### 2.1.1.3 Blockchain Tier

The second half declares the function to be called (**get\_all**) and the arguments that are passed. This function is implemented in the chaincode (**\Chaincode\vehicle\_code\vehicles.go**):

```
func (t *Chaincode) get_all(stub *shim.ChaincodeStub, v Vehicle, current_owner
string, caller_name string, caller_role int64) ([]byte, error) {

    bytes, err := json.Marshal(v)

    if err != nil { return nil, errors.New("Invalid vehicle object") }

    if current_owner == caller_name || caller_role == 0 {

        return bytes, nil
    } else {
        return nil, errors.New("Permission Denied")
    }
}
```

The chaincode function, which is written in the Go programming language<sup>2</sup>, receives the vehicle, calling user, current owner and the callers role which are retrieved by the Query function of the chaincode. This Query function calls out to the CA to get the ecert of the caller and check they are a user of the system and then gets the role from that ecert. The function checks to see if the user either


<sup>2</sup> <https://golang.org/>

owns the car or is has the role of regulator in which case they can see the car. Since the enrolment certificate is stored on, and retrieved from, the Blockchain, it is treated as a reliable source of enrollment and identity information.

On completion of this function, a list of V5C documents is returned to the call (from the user interface tier).

## 2.1.2 Select Dealership

A list of dealerships are similarly displayed in order to transfer the selected vehicle.

Clicking the  icon under Dealership in **manufacturer.html** calls the following function located in **recipient\_popup\_functions.js**:

```
function() {  
    ...  
    recipients = loadRecipients();  
}
```

The **loadRecipients** function is defined in **participant\_functions/loadRecipients** and is responsible for retrieving the names of all dealerships from the Blockchain and formatting the response. The names are retrieved by sending a REST GET request to the Blockchain Demo API:

```
url: '/blockchain/participants/'+newRecPlural.toLowerCase(),
```

The variable **newRecPlural** represents the role of the intended recipient: in this instance, it is dealerships. The Blockchain Demo API processes this request with the following function defined in **router.js**:

```
app.get('/blockchain/participants', function(req,res)  
{  
    participants.read(res)  
})
```

The **read** function is implemented in **participants\CRUD\read.js**. This function loads and returns a predefined list of dealerships that are loaded locally, rather than from the Blockchain<sup>3</sup>. This information is then formatted by the **read** function and returned to the calling function to be displayed in a list on screen.

---

<sup>3</sup> The decision to load dealerships from a local cache was made to simplify the Blockchain for the demo. Indeed, implementations may store dealerships on the Blockchain.

## 2.1.3 Transfer Asset

Interactions between components in the three tier PoT architecture are summarized in Figure 2.

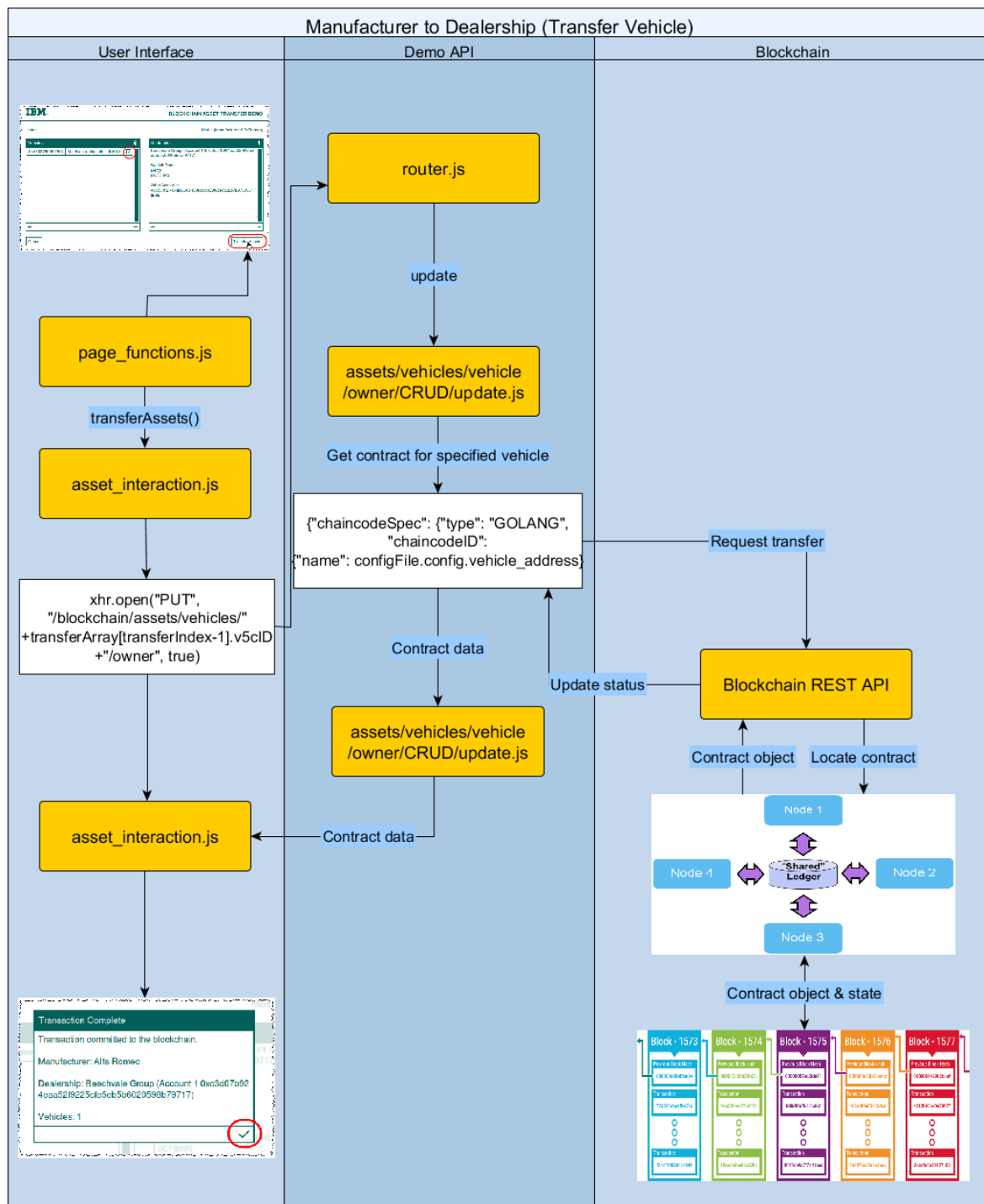


Figure 2 -- Asset Transfer Scenario (Transfer Vehicle)

Clicking the “**Transfer Assets**” button defined in **manufacturer.html** calls the following function location in **page\_functions.js**:

```
function() {
...
  if ($('#selVhclsTbl tr').length >= 1 && $('#recipientInfo').html() != "") {

    var spans = '';
...

    var carDets = [];

    $('#selVhclsTbl tr').each(function() {

      var v5cID = $(this).find('.v5cID').val();
      if (index == $('#selVhclsTbl tr').length) {
        last = true;
      }

      $('#chooseConfHd').html('<span>Transaction Complete</span>');
...

      var data = {};
      data.function_name = transferName;
      data.value = $('#accAddr').html();
      data.v5cID = v5cID;

      carDets.push(data);

    });
    transferAssets(carDets)

  } else if (!($('#selVhclsTbl tr').length <= 1)) {
    $('#failTransfer').show();
    $('#failTxt').html('You have not selected any vehicles to transfer.');
```

This function is responsible for preparing a request to transfer a V5C document from manufacturer to dealership. It begins by ensuring that there is at least one vehicle selected to transfer and a dealership to receive the vehicle.

Then the address of the V5C document (for each vehicle to transfer) is stored in **v5cID** and a data structure is created (called **data**) that stores a function name (“*manufacturer\_to\_private*”) along with the name of the recipient (in **value**) and the V5C identity that was previously collected. This data structure is then stored in an array of vehicle details and is passed as an argument to the **transferAssets** function, which is defined in **asset\_interaction.js**.

This function calls **transferAsset** which sends a request (REST PUT) to the Blockchain Demo API and includes the address of the V5C contract:

```
xhr.open("PUT", "/blockchain/assets/vehicles/"+transferArray[transferIndex-1].v5cID+"/owner", true)
```

This invokes the following function, defined in **router.js**:

```
app.put('/blockchain/assets/vehicles/:v5cID/owner' , function(req,res)
{
  vehicle.owner.update(req,res)
});
```

The **update** function is implemented in **assets/vehicles/vehicle/owner/CRUD/update.js**. This function constructs a request for updating the owner of a vehicle / V5C contract:

```
var chaincodeInvocationSpec = {
    "chaincodeSpec": {
        "type": "GOLANG",
        "chaincodeID": {
            "name": configFile.config.vehicle_address
        },
        "ctorMsg": {
            "function": function_name.toString(),
            "args": [
                req.session.user,
                newValue.toString(),
                v5cID
            ]
        },
        "secureContext": req.session.user,
        "confidentialityLevel": "PUBLIC"
    }
}

var options = {
    url: configFile.config.api_url+'/devops/invoke',
    method: "POST",
    body: chaincodeInvocationSpec,
    json: true
}
```

As before the HTTP Request is given the details of the chaincode that will be invoked, the function to call and the arguments to pass to that function. The REST call uses the URL of one of the peers.

```
request(options, function(error, response, body){
    if (!error && response.statusCode == 200)
    {
        ...
        var url = http://localhost/blockchain/assets/vehicles/'+v5cID+'/owner';
        ...
        var options = {
            url: url,
            method: 'GET',
            jar: j
        }
        ...
        var counter = 0;
        var interval = setInterval(function(){
            if(counter < 5){
                request(options, function (error, response, body) {
                    if (!error && response.statusCode == 200)
                    {
                        ...
                        res.end(JSON.stringify(result))
                        clearInterval(interval);
                    }
                })
            }
        }, 1500)
    }
    ...
})
```

The code above calls out to the read function and passing the new owner as a cookie. If the new owner can read the car then the server side knows the transfer was a success.

In the chaincode (which is implemented in the Blockchain API), various checks are made to establish the validity of the transaction before transferring ownership from manufacturer to dealership such as

the vehicle being fully defined. Finally, a transfer is logged to the Blockchain via the vehicle\_logs chaincode:

```
func (t *Chaincode) manufacturer_to_private(stub *shim.ChaincodeStub, v Vehicle,
current_owner string, caller_name string, caller_ecert []byte, caller_role int64,
recipient_name string, recipient_ecert []byte, recipient_role int64) ([]byte,
error) {
    if v.Make == "UNDEFINED" ||
        v.Model == "UNDEFINED" ||
        v.Reg == "UNDEFINED" ||
        v.Colour == "UNDEFINED" ||
        v.VIN == 0 {
        return nil, errors.New("Car not fully defined")
    }
    if v.Status == STATE_MANUFACTURE &&
        current_owner == caller_name &&
        caller_role == ROLE_MANUFACTURER &&
        recipient_role == ROLE_PRIVATE_ENTITY &&
        v.Scrapped == false {
        v.Owner = string(recipient_ecert)
        v.Status = STATE_PRIVATE_OWNERSHIP
    } else {
        return nil, errors.New("Permission denied")
    }

    _, err := t.save_changes(stub, v)
    if err != nil { return nil, err }

    _, err = t.create_log(stub, []string{ "Transfer",
        caller_name + " → " + recipient_name +
        "&&[" + strconv.Itoa(v.VIN) + "]" +
        v.Make + " " + v.Model + ", " +
        v.Colour + ", " + v.Reg,
        v.V5cID, caller_name, recipient_name})
    if err != nil { return nil, err }

    return nil, nil
}
```

**Scenario: Verify Transaction activity using Regulator View**

To recap, in this scenario the Regulator will view the asset transfer and disposal activity. Unlike other participants, the Regulator has an unrestricted view of all activities recorded on the Blockchain.

Selecting the 'Regulator View' option from the main menu opens the **regulator-view.html** page. This page initiates a script called **ledger.js**:

```
<script src="JavaScript/ledger/ledger.js" type="text/javascript"></script>
```

This script begins by retrieving all blocks available on the Blockchain by sending a REST request to the Demo API:

```
$.ajax({
  type: 'GET',
  dataType: 'json',
  contentType: 'application/json',
  crossDomain: true,
  url: '/blockchain/events',
  success: function(d) {
    formatEvents(d.Result)
  },
  ...
})
```

The **formatEvents** function iterates through the response and formats the page with the retrieved activity information. Much of the remainder of **ledger.js** is dedicated to formatting and filtering this information on page.

An HTTP Request calls to the API and is given the details of the chaincode that will be queried, the function to call and the arguments to pass to that function. The REST call uses the URL of one of the peers.

```
var chaincodeInvocationSpec = {
  "chaincodeSpec": {
    "type": "GOLANG",
    "chaincodeID": {
      "name": configFile.config.event_address
    },
    "ctorMsg": {
      "function": "get_vehicle_logs",
      "args": [req.session.user]
    },
    "secureContext": req.session.user,
    "confidentialityLevel": "PUBLIC"
  }
}

var options = {
  url: configFile.config.api_url+'/devops/query',
  method: "POST",
  body: chaincodeInvocationSpec,
  json: true
}

request(options, function(error, response, body)
{
  if (!error && response.statusCode == 200)
  {
    ...
  }
})
```



The identity of the appropriate chaincode is given and the function defined on that chaincode is declared (**get\_vehicle\_logs**).

The Blockchain API implements the **get\_vehicle\_logs** method (in **Chaincode/vehicle\_log\_code**) as:

```
func (t *Chaincode) get_vehicle_logs(stub *shim.ChaincodeStub, args []string)
([]byte, error) {
    bytes, err := stub.GetState("Vehicle_Logs")

    ...

    var eh Vehicle_Log_Holder

    err = json.Unmarshal(bytes, &eh)

    ...

    ecert, err := t.get_ecert(stub, args[0])

    ...

    role, err := t.check_role(stub, []string{string(ecert)})

    ...

    if role == ROLE_AUTHORITY {
        repNull := strings.Replace(string(bytes), "null", "[]", 1)
        return []byte(repNull), nil
    } else {
        return t.get_users_vehicle_logs(stub, eh, args[0])
    }
}
```

The code gets from the blockchain the vehicle\_logs which are stored in a json format. It then converts that into a go structure. After that it gets the ecert of the user and if that ecert has the role of being an authority then all the vehicle logs are returned (the regulator view uses a user with the role of an authority). Otherwise it gets just the events the user is allowed to see e.g. what happened to cars they own/owned before they owned them. Once all the events have been retrieved they are returned to the user interface tier.

This completes the process of retrieving transactions for the Regulator view. Note that no restrictions are placed on access to the Blockchain: the Regulator sees all transactions that are available on the Blockchain. This ensures a consistent and complete view for the purposes of meeting regulatory audit requirements.

## 2.2 Scenario: Create Vehicle Template (Regulator)

To recap, in this scenario a Vehicle Template is created by the Regulator to be used by the Manufacturer to identify a vehicle.

Selecting the 'Create Asset' option from the main menu opens the **regulator.html** page. On this page, clicking the 'Create V5C' button invokes the **createAsset** function that is defined in **asset\_functions/asset\_interaction.js**:

```
<script src="JavaScript/asset_functions/asset_interaction.js"
type="text/javascript"></script>
<span id="createV5C" class="lftBtn mnBtn" onclick="createAsset()">Create V5C</span>
```

The **createAsset** function calls the Demo API, implemented in **router.js**, with a POST REST call to create a new V5C document:

```
xhr.open("POST", "/blockchain/assets/vehicles/", true)
```

The Demo API in turn receives the request and calls the **createV5cID** function defined in **assets/vehicles/CRUD/create.js**:

The **createV5cID** function generates a v5cID then calls the **checkIfAlreadyExists** to ensure that it is unique.

The **checkIfAlreadyExists** function queries the Blockchain API and requests the details of a specific vehicle:

```
var chaincodeInvocationSpec = {
  "chaincodeSpec": {
    "type": "GOLANG",
    "chaincodeID": {
      "name": configFile.config.vehicle_address
    },
    "ctorMsg": {
      "function": "get_all",
      "args": [req.session.user, v5cID]
    },
    "secureContext": req.session.user,
    "confidentialityLevel": "PUBLIC"
  }
};

var options = {
  url: configFile.config.api_url+'/devops/query',
  method: "POST",
  body: chaincodeInvocationSpec,
  json: true
}

request(options, function(error, response, body)
{
  if (!error && response.statusCode == 200)
  {
    createV5cID(req, res)
  }
  else
  {
    ...
    createVehicle(req, res, v5cID)
  }
})
```

If the call is successful then it means a car exists so it goes back and calls **createV5cID** to generate a new one and try again. Otherwise, the **createVehicle** function is called, which calls the **create\_vehicle** function on the Blockchain API:

```
var chaincodeInvocationSpec = {
  "chaincodeSpec": {
    "type": "GOLANG",
    "chaincodeID": {
      "name": configFile.config.vehicle_address
    },
    "ctorMsg": {
      "function": "create_vehicle",
      "args": [
        req.session.user, v5cID
      ]
    },
    "secureContext": req.session.user,
    "confidentialityLevel": "PUBLIC"
  }
};

var options = {
  url: configFile.config.api_url+'/devops/invoke',
  method: "POST",
  body: chaincodeInvocationSpec,
  json: true
}
```

```
request(options, function(error, response, body){
  if (!error && response.statusCode == 200) {
    ...
  }
})
```

In turn, the **create\_vehicle** function, implemented in **Chaincode/vehicle\_code/vehicles.go**, validates the request for a new vehicle (and the authority of the user requesting this) and if all checks pass, will write the vehicle template to the ledger and call the **create\_log** function:

```
v5c_ID      := "\"v5cID\": \""+v5cID+"\", "
vin         := "\"VIN\":0, "
make        := "\"Make\": \"UNDEFINED\", "
model       := "\"Model\": \"UNDEFINED\", "
reg         := "\"Reg\": \"UNDEFINED\", "
owner       := "\"Owner\": \""+string(caller_ecert)+"\", "
colour      := "\"Colour\": \"UNDEFINED\", "
leaseContract := "\"LeaseContractID\": \"UNDEFINED\", "
status      := "\"Status\":0, "
scrapped    := "\"Scrapped\":false"

vehicle_json :=
"{"+v5c_ID+vin+make+model+reg+owner+colour+leaseContract+status+scrapped+"}"#

...

err = json.Unmarshal([]byte(vehicle_json), &v)

...

_, err = t.save_changes(stub, v)
_, err = t.create_log(stub, []string{ "Create",
                                      "Create V5C",
                                      v.V5cID, caller_name})

...
```

Once complete, the new V5C contract exists in the Blockchain and is available to be transferred to the manufacturer.

## 2.3 Scenario: View Blockchain Activity

To recap, the Blockchain activity can be reviewed by selecting the 'Live Stats' link on the main menu. This view provides real-time information of the running Blockchain and can be used to confirm the current state of the Blockchain.

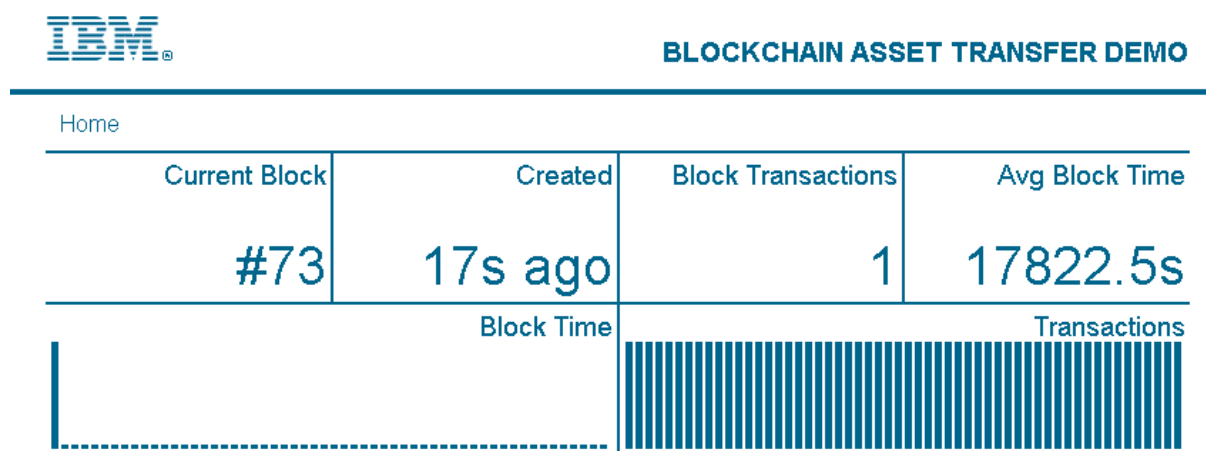


Figure 3 -- View of Blockchain Activity

The parts of the activity view have the following meaning:

**Current Block** is the number of the most recent block added to the chain.

**Created** is the time elapsed since the last block was added to the chain.

**Block Transactions** is the number of transactions that were packed into the most recent block<sup>4</sup>.

**Avg Block Time** displays the average time currently taken to add a block.

**Block Time** shows a graph of the time taken for each of the blocks

**Transactions** indicates when transaction-related functions are called, such as during an asset transfer.

The **stats.html** page is responsible for displaying the real-time Blockchain activity. This page instantiates the following two Javascript files:

```
<script src="JavaScript/charts.js" type="text/javascript"></script>
<script src="JavaScript/stats.js" type="text/javascript"></script>
```

The first, **charts.js**, provides the formatting required to arrange the activity data on the page. The second, **stats.js**, retrieves the activity data and computes the values that are required to be displayed. All values for the stats page are collected from the following request URLs:

```
url: '/blockchain/blocks',
url: '/blockchain/blocks/' + (blockNum),
url: '/blockchain/blocks/' + (blockNum - i),
```

All requests are made from **stats.js** and the results are stored in the following elements:

```
#currBlock
#timeSince
#avgTime
#avgTime
#transLast
```

<sup>4</sup> This number tends to remain at '1' due to the nature and speed of the block consensus process in the demonstration.

The refresh time of the page (once every 10 seconds to prevent overwhelming the peer) is also defined in **stats.js**:

```
window.setInterval(updatePage, 10000);
```

The requester function (another HTTP Request) for the Demo API calling to get a specific block is implemented in **Blockchain/blocks/block/CRUD/read.js**:

```
var options = {
    url: configFile.config.api_url+'/chain/blocks/'+req.params.blockNum,
    method: "GET"
};
request(options, function (error, response, body){
    if (!error && response.statusCode == 200) {
    ...
    }
```

The Blockchain API provides several methods for querying the state of the Blockchain, including:

```
/chain
/chain/blocks/{Block}
/transactions/{UUID}
```



## Section 3. Chaincode in Blockchain

### 3.1 Chaincode Overview

In IBM Blockchain, we use chaincode to implement Smart Contracts on the blockchain.

We use a single Vehicle chaincode as a way to interact with all vehicles on the blockchain, we also use a Vehicle\_Log chaincode which stores all the information about any interactions with any other vehicles on the blockchain.

Both the chaincodes are written in Go, this means we are able to include added functionality like callouts to other systems.

We will go into more detail about what is included in these chaincodes and how they work in the next sections.

There are 3 ways to interact with a chaincode using the IBM Blockchain API

- **Deploy** – This is used when you want to deploy a new chaincode to blockchain, once a chaincode has been deployed a chaincode ID will be returned which is what is required to perform invoke and query functions on this chaincode. This is a transaction which once validated through the consensus method, will be added to the blockchain.
- **Invoke** – This is for all functions where data will be changed in some way, also you aren't able to return anything using invoke. Calling an invoke function in the chaincode will create a transaction which will be added to the blockchain.
- **Query** – This is for when you call a function which retrieves some information stored in a chaincode and return it to the user, this is for read only operations. E.g. used when we want to retrieve a vehicles details.

All chaincode has to have a Run and a Query function, as defined by IBM Blockchain.

The Run function is what gets called when you want to deploy a chaincode and every time there is an invoke on a chaincode, this function will then call the relevant function within the chaincode depending on the value of the parameter "function". An example Run function is shown below (taken from the chaincode Vehicle\_Log):

```
func (t *Chaincode) Run(stub *shim.ChaincodeStub, function string, args []string)
([]byte, error) {
    if function == "init" { return t.init(stub)
    } else if function == "create_log" {
        if(len(args) < 4) {
            return nil, errors.New("Invalid number of arguments supplied")
        }
        var users_involved []string

        for i := 3; i < len(args); i++ {
            users_involved = append(users_involved, args[i])
        }
        return t.create_log(stub, args[0], args[1], args[2], users_involved)
    }
    return nil, errors.New("Function of that name not found")
}
```

The Query function does the same as the Run function as explained above except it handles all query requests on the chaincode. It handles the request in the same way as the Run function, routing the request to the correct function depending on the "function" parameter passed using the IBM Blockchain API. Here is an example Query function (taken from the Vehicle\_Log chaincode):

```
func (t *Chaincode) Query(stub *shim.ChaincodeStub, function string, args []string)
([]byte, error) {

    if function == "get_logs" { return t.get_logs(stub, args) }

    return nil, errors.New("Function of that name not found")
}
```

## 3.2 Vehicle Chaincode

The Vehicle Chaincode is used to interact with vehicles which are on the blockchain, through this chaincode you are able create, transfer, update, scrap and read vehicles.

The data structure is also defined within this contract and is defined as follows:

```
type Vehicle struct {

    Make          string `json:"make"`
    Model         string `json:"model"`
    Reg           string `json:"reg"`
    VIN           int    `json:"VIN"`
    Owner        string `json:"owner"`
    Scrapped      bool   `json:"scrapped"`
    Status        int    `json:"status"`
    Colour        string `json:"colour"`
    V5cID         string `json:"v5cID"`
    LeaseContractID string `json:"leaseContractID"`

}
```

Each car added to the blockchain is an instance of the struct **Vehicle**. This struct stores the details of a car in a JSON object.



## 3.3 Vehicle Functions

### 3.3.1 Constructor

When the chaincode is initially deployed to the blockchain the function called is `init`. This is a very simple function which initialises the chaincode and doesn't return anything. The function code is shown below:

```
func (t *Chaincode) init(stub *shim.ChaincodeStub, args []string) ([]byte, error) {
    return nil, nil
}
```

### 3.3.2 Validation functions

In IBM Blockchain, membership services are included as a part of the fabric. A user has to register with the Certificate Authority (CA) with their username and password and if the registration is successful, a certificate called an eCert is created which contains the user's information.

This means we are able to retrieve information about a user through the CA by getting their eCert. This then enables us to validate any user involved in a function call to determine if they have permission to do anything. The functions which allow us to do this are `get_ecert` and `check_role`, they are both explained below.

#### 3.3.2.1 GET\_ECERT

Defined in the chaincode as:

```
func(t *SimpleChaincode) get_ecert(stub *shim.ChaincodeStub, args []string)
([]byte, error)
```

This function retrieves the eCert of a certain user by performing a HTTP Get using the IBM Blockchain API, as shown below:

```
response, err :=
http.Get("http://localhost:5000/registrar/"+args[0]+"/ecert")
```

`args[0]` is the username of the user whose eCert we want to get.

The function then does some error checking to ensure the response is in the format we expect, then returns the result as `[]byte`.

### 3.3.2.2 CHECK\_ROLE

Defined in the chaincode as:

```
func(t *SimpleChaincode)check_role(stub *shim.ChaincodeStub, args []string)
([]byte, error)
```

This function is used to check that a user's role, which is stored in their eCert, is the correct value for what they are trying to do. E.g. check this user is a Manufacturer.

`check_role` takes the eCert which was retrieved by the function `get_ecert` and decodes and converts the certificate to X509 format. This process is shown below:

```
decodedCert, err := url.QueryUnescape(args[0])

if err != nil {

return nil, errors.New("Could not decode certificate")

}

pem, _ := pem.Decode([]byte(decodedCert))

x509Cert, err := x509.ParseCertificate(pem.Bytes)

if err != nil {

return nil, errors.New("Couldn't parse certificate")

}
```

`args[0]` is the eCert that has been passed.

Once this has been completed, the function will look up the role value stored in the eCert and compare this role value to the value passed, role is stored in the eCert as an integer.

If the role value in the eCert and the passed role value are equal then the function returns the string "true", otherwise it will return the string "false".

### 3.3.3 Invoke functions

These functions are used for when you want to change any data within the chaincode. For the demo, we use invoke functions to create, transfer, update and scrap vehicles.

#### 3.3.3.1 CREATE\_VEHICLE

Defined in the chaincode as:

```
func (t *Chaincode) create_vehicle(stub *shim.ChaincodeStub, caller_name string,
    caller_ecert []byte, caller_role int64, v5cID string) ([]byte, error)
```

Here we can see what inputs are required and the data types returned, `stub *shim.ChaincodeStub` is defined by IBM Blockchain and is what allows us to store information and retrieve information in the ledger. Examples are shown below:

```
record, err := stub.GetState(v.V5cID)

if record != nil {

    fmt.Println("ERROR: Vehicle already exists")
    return nil, errors.New("Vehicle already exists")
}
```

This is used to retrieve information stored in the ledger with the key `v.V5cID`, we do this to check that a vehicle doesn't already exist with the same `v5cID`, ensuring we don't overwrite any information.

```
err = stub.PutState(v.V5cID, bytes)

if err != nil {
    return nil, err
}
```

This is how we store information about the newly created vehicle. It is stored in the ledger with a unique `v5cID` as a key so we are able to retrieve the vehicle's information at a later date.

By using the functions `get_ecert` and `check_role`, which are explained above, we are able to ensure that only an Authority is able to create a vehicle, if a user who isn't an Authority tries to create a vehicle, the vehicle won't be created and we return the error "Permission Denied", shown below:

```
if isAuth {
    err = stub.PutState(v.V5cID, bytes)

    if err != nil {
        return nil, err
    }
} else {
    return nil, errors.New("Permission Denied")
}
```

`isAuth` is the value returned from `check_role` when the passed role value is "0", which is what we assign as the role for authorities.

### 3.3.3.2 AUTHORITY\_TO\_MANUFACTURER

Defined in the chaincode as:

```
func (t*SimpleChaincode)authority_to_manufacturer(stub *shim.ChaincodeStub, args
[]string) ([]byte, error)
```

When we retrieve a vehicle from the ledger we need to store this information in a Vehicle object, which is defined within the chaincode, so that we can interact with the vehicle as a JSON object. This makes it simpler for us to read and update specific attributes. This process is shown below:

```
var v Vehicle

...

bytes, err = stub.GetState(args[2])

if err != nil {
    return nil, errors.New("Error retrieving vehicle with v5cID = " +
args[2])
}

err = json.Unmarshal(bytes, &v)
```

`args[2]` is the v5cID of the vehicle we want to get from the ledger.

Before we can transfer a vehicle from an authority to a manufacturer we need to check that all the passed information is valid. This means we have to retrieve and convert the eCerts of both the sender and receiver. This is done by using the `get_ecert` and `check_role` functions, which have been explained above.

For a transfer from authority to manufacturer we need to check that the sender is an authority and the recipient is a manufacturer, this is where we use `get_ecert` and `check_role`, shown below is how we use these to determine if the sender is the correct role (process is the same for checking the recipient's role):

```
caller_ecert, err := t.get_ecert(stub, []string{args[0]})

if err != nil {
    return nil, errors.New("Could not find ecert for user: "+args[0])
}

owner_role, err := t.check_role(stub, []string{string(caller_ecert), "0"})

isAuth,err := strconv.ParseBool(string(owner_role))
```

The conditions which need to be met for a transfer from an authority to a manufacturer is shown below:

```
if v.Owner == string(caller_ecert) &&
```

```

v.Status == 0 &&
isAuth &&
isMan &&
!v.Scrapped {

    v.Owner = string(rec_ecert)
    v.Status = 1

} else {

    return nil, errors.New("Permission Denied")

}

```

Above you can see that we check the user who called this function is the owner of the vehicle, the vehicle has the correct status value, the sender is an authority, the recipient is a manufacturer and the vehicle hasn't been scrapped.

Only if all these conditions have been met do we update the owner attribute in the vehicle to be the recipient, we also update the status attribute to reflect the change in the vehicle's lifecycle. Once updated, the new vehicle is stored in the ledger by doing:

```
err = stub.PutState(v.V5cID, bytes)
```

The other transfer functions work in the same way by checking if the executer is the owner and if both the executer and recipient are of the right type for the function call and validating the car's status. The status is set by them to show which state the car is currently in.

### 3.3.3.3 UPDATE\_VIN

Defined in the chaincode as:

```

func (t*SimpleChaincode)update_vin(stub *shim.ChaincodeStub, args []string)
([]byte, error)

```

This function, and all other update functions, follows the same structure as the transfer functions where the general process is:

- Retrieve vehicle from the ledger
- Validation on vehicle and user details
- Update relevant vehicle attributes
- Store updated vehicle in the ledger

The retrieval of a vehicle and converting it into a JSON object is done the same way as above in the `authority_to_manufacturer` function.

The new VIN value the user wants to update the vehicle with is converted from a string to an integer, as defined in the **Vehicle** struct. We also do validation on the length of the integer as we require all VINs to be 15 digits in length.

The conditions which need to be met in order to update a vehicle's VIN are shown below:

```

if v.Status == 1 &&
    v.Owner == string(caller_ecert) &&
    isMan &&
    v.VIN == 0 &&

```

```
!v.Scrapped {  
  
    v.VIN = newVin  
  
} else {  
    return nil, errors.New("Permission denied")  
}
```

For a user to update the VIN, the vehicle has to have a status of 1, the user is the owner, the user is a manufacturer, the current VIN value is 0 and the vehicle is not scrapped.

We have the condition of `v.VIN == 0` because we only allow the user to update the VIN once.

Once the VIN has been updated then the updated vehicle is stored in the ledger. Again, this process is the same as in the `authority_to_manufacturer` function.

The other update functions are the same in format to these however, they do not look at whether the asset being changed is unassigned yet as the make, model, colour and registration can be updated as often as needed by the manufacturer, with dealerships and leasees able to update colour and registration only.

The other update function is for the scrap merchant to update scrapping: this is the same and it checks if they are a scrap merchant, the owner and the vehicle hasn't been scrapped before changing the scrapped value, we check a vehicle hasn't been scrapped before to prevent multiple scraps of the same vehicle.

### 3.3.4 Query functions

We use query functions when we need to return information stored in the chaincode to the user. The only function we use which is a query is used to get all the details of the vehicle and return this to the user.

#### 3.3.4.1 GET\_ALL

Defined in the chaincode as:

```
func (t*SimpleChaincode)get_all(stub *shim.ChaincodeStub, args []string) ([]byte, error)
```

This function retrieves the vehicle from the ledger and returns this information to the user who called it, providing they have permission to view the vehicle.

Retrieving the vehicle is the same process as in the `authority_to_manufacturer` function. The conditions which are required to be met to get a vehicles details are shown below:

```
if string(caller_ecert) == v.Owner ||
    isAuth {

    fmt.Printf("Query Response:%s\n", v)

    return bytes, nil

} else {
    return nil, errors.New("Permission denied")
}
```

To be able to read a vehicle's details, you must either be the owner of the vehicle or an authority. `v` is the vehicle which was retrieved from the ledger converted to a JSON format.

### 3.3.5 Chaincode invoking chaincode

For every create, transfer, update and scrap of a vehicle we create a log of this happening. This is so that we can quickly view all interactions for a vehicle and it is what we use to populate the regulator view.

We add and store logs by using another chaincode called "vehicle\_log" and all invocations of this chaincode are done through the vehicle chaincode, once all permissions have been checked and the attributes updated. Here is the function in the vehicle chaincode which invokes the vehicle\_log chaincode:

```
func (t *Chaincode) create_log(stub *shim.ChaincodeStub, args []string) ([]byte, error) {

    chaincode_name:=
    "6e266ca5c0a1384b8ae722dd1d94a726d269fccccfcfb18cf4d3e1984f20805aeff2eaffa111b835ebe
    468d1cd9e9785c3bef08418921efbdc51bbe1b3837a129"

    chaincode_function := "create_log"

    chaincode_arguments := args

    _, err := stub.InvokeChaincode(chaincode_name, chaincode_function,
    chaincode_arguments)
```

```

        if err != nil { return nil, errors.New("Failed to invoke vehicle_log_code")
    }

    return nil,nil
}

```

`chaincode_name` is the unique identifier of the chaincode we want to invoke, this is generated by IBM Blockchain and returned when you deploy the chaincode.

`chaincode_function` is the function in `vehicle_log` we want to call when we invoke the `vehicle_log` chaincode.

`chaincode_arguments` is the array of arguments we want to pass.

`stub.InvokeChaincode` here we use `stub` which is of type `*shim.ChaincodeStub` which we use so that we can invoke another chaincode from a chaincode.

Below is an example of this function actually being used in the `vehicle` chaincode:

```

_, err = t.create_log(stub,[]string{
    "Transfer",
    caller_name + " → " + recipient_name + "&&[" + strconv.Itoa(v.VIN) + "]" +
    v.Make + " " + v.Model + ", " + v.Colour + ", " + v.Reg,
    v.V5cID, caller_name, recipient_name})

```

The first parameter passed is the type of interaction which has just happened, in this case it is a `"Transfer"`.

The next parameter is the details of the interaction, for a transfer we show who went from and to and the vehicle details.

We also pass the V5C ID of the vehicle and the users involved.

Below is an explanation of how the `Vehicle_log` chaincode works.

## 3.4 Vehicle\_log Chaincode

The `Vehicle_log` Chaincode is used to allow the adding and reading of logs. We create a new log a result of some interaction happening with a vehicle in the `vehicle` chaincode. The `vehicle` chaincode invokes this chaincode to create a new log.

The data structure for a log is also defined within this contract and is defined as follows:

```

type Log struct {
    Name      string `json:"name"`
    Time      string `json:"time"`
    Text      string `json:"text"`
    Obj_ID    string `json:"obj_id"`
    Users     []string `json:"users"`
}

```

Each log added uses the struct **Log**. This struct is a JSON object which stores the details of a log which we require to show everything that has happened on the blockchain in the regulator view.

We also have a struct called **LogsHolder** and this is simply an array of logs objects, this is where all new logs are added to before being added to the ledger.



## 3.5 Vehicle\_log Functions

### 3.5.1 Constructor

The constructor for the vehicle\_log chaincode is very simple. All it does is create an instance of LogsHolder, which initially contains no logs, and adds it to the ledger. The function code is shown below:

```
func (t *Chaincode) init(stub *shim.ChaincodeStub) ([]byte, error) {

    var eh LogsHolder

    bytes, err := json.Marshal(eh)
    if err != nil { return nil, errors.New("Error creating log record") }

    err = stub.PutState("Vehicle_Log", bytes)
    if err != nil { return nil, errors.New("Error creating blank logs array") }

    return nil, nil
}
```

Here you see the use of `stub.PutState` again, as this is what is used to add something to the ledger. We use "Vehicle\_Log" as the key and we are able to get this information back from the ledger later by providing this key when we use `stub.GetState`.

### 3.5.2 Validation Functions

The vehicle\_log chaincode also makes use of the same validation functions as the vehicle chaincode, these are `get_ecert` and `check_role`. For more information on these, they are explained in the section "Vehicle Functions", under "Validation Functions".

### 3.5.3 Invoke Functions

#### 3.5.3.1 CREATE\_LOG

The function is defined as follows:

```
func (t *Chaincode) create_log (stub *shim.ChaincodeStub, log_name string, log_text
string, log_obj_id string, log_users []string) ([]byte, error)
```

This is the only other function which is called by using an invoke on the chaincode vehicle\_log. This is the function to create and then add a log to LogsHolder which then gets added to the ledger.

`create_log` also formats the passed information and stores it in a logs object, shown below:

```
var e Log

e.Name      = log_name
e.Time      = time.Now().Format("02/01/2006 15:04")
e.Text      = log_text
e.Obj_ID    = log_obj_id
e.Users     = log_users
```

Once all the information required for a log has been added, we retrieve the LogsHolder object from the ledger by using the key "Vehicle\_Log" and convert this to a json object.

```
bytes, err := stub.GetState("Vehicle_Log")

if err != nil { return nil, errors.New("Unable to get logs") }

var eh LogsHolder

err = json.Unmarshal(bytes, &eh)
```

The Log object `e` then gets appended to the LogsHolder object `eh` and this then gets added to the ledger using `PutState`.

```
eh.Logs = append(eh.Logs, e)

bytes, err = json.Marshal(eh)

if err != nil { fmt.Print("Error creating log record") }

err = stub.PutState("Vehicle_Log", bytes)
```

As said before this function is only invoked from the Vehicle chaincode and only once all the user and passed information has been validated and the relevant vehicle information has been updated. This ensures there are no logs added for things which didn't happen.

## 3.5.4 Query functions

We want to query the logs so that we can quickly information about what has happened on the blockchain.

This is used in a few areas in the demo, the first is to populate the regulator view and shows everything that has happened to any of the vehicles on the blockchain. The second is for when we want to retrieve all the vehicle information for a specific owner, this is used, for example, when you want to select vehicles to transfer and ensures no vehicles appear which that participant doesn't own.

### 3.5.4.1 GET\_LOGS

The function is defined as follows:

```
func (t *Chaincode) get_logs(stub *shim.ChaincodeStub, args []string) ([]byte, error)
```

This is the only query function you can call directly from the IBM Blockchain API, all other query functions in the chaincode are called as a result of the information passed to this function.

This function is fairly simple in that it retrieves the LogsHolder object from the ledger and converts it to a JSON object.

```
bytes, err := stub.GetState("Vehicle_Log")

if err != nil { return nil, errors.New("Unable to get logs") }

var eh LogsHolder

err = json.Unmarshal(bytes, &eh)
```

It will then retrieve the ecert of the user which was passed to this function and if they are an authority it will return the whole LogsHolder object as an authority can see everything that has happened on the blockchain.

```
ecert, err := t.get_ecert(stub, args[0])

if err != nil {    return nil, err }

role, err := t.check_role(stub, []string{string(ecert)})
if err != nil { return nil, err }

if role == ROLE_AUTHORITY {                // Return all logs if authority

    repNull := strings.Replace(string(bytes), "null", "[]", 1)
    // If the array is blank it has the json value null so we need to make it an empty
    array

    return []byte(repNull), nil
}
```

If the user isn't an authority, another function gets called which we will go into more detail below.

### 3.5.4.2 GET\_USERS\_LOGS

The function is defined below:

```
func (t *Chaincode) get_users_logs(stub *shim.ChaincodeStub, eh LogsHolder, name
string) ([]byte, error)
```

This function will return a subset of the logs so that only things the user is involved will be included, this means that the user is only able to see logs relevant to them and also means retrieving all vehicles owned by them is much quicker.

The function does this by looking at each Log object in the LogsHolder object and checks if their username appears in the Users attribute in the Logs object. This is shown below:

```
for i, log := range eh.Logs {

    if contains(log.Users, name) {

        users_logs = append(users_logs, log)

        users_logs = append(users_logs, t.get_obj_logs(stub, eh.Logs,
log.Obj_ID, searched_to[log.Obj_ID], i, name)...)
        // get the logs of the car before the user had ownership

        searched_to[log.Obj_ID] = i;
    }
}
```

## End Of Lab