Implementing blockchain for cognitive IoT applications, Part 2: Use vehicle sensor data to execute smart transactions in Blockchain

Integrate the IBM Watson IoT Platform with IBM Blockchain services to develop a sample use case

Amitranjan Gantait
Joy Patra
Ayan Mukherjee

June 05, 2017

Start implementing blockchain for cognitive IoT applications. Part 1 of this two-part series discussed the use of blockchain in IoT solutions and explored how different industries are leveraging these two technologies to build end-to-end automated and secured solutions. Part 2 shows you how to use the IBM Watson IoT platform and IBM Blockchain service to build a sample use case.

In Part 1, we discussed the use of blockchain in IoT solutions and explored how different industries are leveraging these two technologies to build end-to-end automated and secured solutions. In Part 2, we will show you how to use the IBM Watson IoT platform and IBM Blockchain service to build a sample use case.

Use case details

This scenario focuses primarily on vehicle data such as Vehicle to Vehicle (V2V) messages, Vehicle to Infrastructure (V2I) messages, and onboard vehicle monitoring data. Therefore, typical attributes include timestamp, latitude, longitude, elevation, speed, direction vector, and acceleration of the connected vehicle and nearby vehicles. Although not included in this demo, additional onboard data for the managed vehicle, such as fuel level and oil levels, as well as other
supporting data (weather information, road conditions, location-specific driving regulations), can be leveraged to extend the use case even further.

“It is relatively easy to quickly build a cognitive IoT application that integrates with an IBM Blockchain service instance.”

Table 1 describes the participants for our scenario.

**Table 1. Key participants**

<table>
<thead>
<tr>
<th>Participant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vehicle</td>
<td>In-car sensors and V2V/V2I sensors send all the related data that's required to execute the transactions in the blockchain.</td>
</tr>
<tr>
<td>Government authority</td>
<td>Typically, the government authority monitors driving patterns (based on specific zones) and sets driver ratings. It may impose penalties based on these driving patterns (such as speeding).</td>
</tr>
<tr>
<td>Driver or owner</td>
<td>Views the driver rating.</td>
</tr>
<tr>
<td>Insurer</td>
<td>Reads driver rating and updates chaincode rules to adjust insurance-related values (such as insurance premium).</td>
</tr>
</tbody>
</table>

The demo application includes a view of the ledger that stores all the interactions that the participants listed above have with the managed vehicles (assets). The Government authority and the Insurer can use the ledger view to review and rate the driving behavior of a specific vehicle and nearby vehicles. Though not included in this demo application, it can also be extended to allow the car service providers to determine vehicle performance and status on the road and trigger maintenance bookings based on the rules that are set up in the blockchain.

Thus, all participants can react based on the data received and trigger actions on the ledger attributes that they own. Some possible scenarios are described in Table 2:

**Table 2. Scenarios**

<table>
<thead>
<tr>
<th>Participant</th>
<th>Scenario</th>
<th>Smart contract action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insurer</td>
<td>Revise insurance rating/premium</td>
<td>If driving rating goes below a certain threshold, revise insurance rating/premium</td>
</tr>
<tr>
<td>Vehicle (IoT)</td>
<td>Record driving pattern or dangerous driving</td>
<td>Update driving rating</td>
</tr>
<tr>
<td>Vehicle (IoT)</td>
<td>Record vehicle parts status</td>
<td>Keep parts status updated based on real-time inputs</td>
</tr>
<tr>
<td>Vehicle (V2V/V2I)</td>
<td>Identify dangerous driving Record traffic violation</td>
<td>Update driving rating Update traffic record</td>
</tr>
<tr>
<td>Government authority</td>
<td>Send traffic case</td>
<td>In case of a violation, create a traffic case May support automated fine based on identified traffic violation</td>
</tr>
</tbody>
</table>

Here are the details for these scenarios:

- Primary asset class:
• **Vehicle**—encompasses vehicle driving data, vehicle parts data, driving rating, insurance record, traffic violation data

• **Events:**
  - **Unsafe driving**—derived from the vehicle IoT data that record location, movement, speed, direction, and so on
  - **Traffic violation or dangerous driving**—recorded from the V2V/V2I data
  - **Insurance rate revised**—derived from the other events such as Unsafe driving and Traffic violation
  - **Traffic case resolved**—when the government authority has resolved the case pertaining to the traffic violation

• **Rules and alerts:**
  - **Traffic case pending** rule raises an alert on a Traffic violation event, until cleared by a Traffic case resolved event
  - **Long history of dangerous driving** could trigger other related actions

Please refer to [Introduction to Hyperledger Smart Contracts for IoT Best Practices and Patterns](https://ibm.com/developerWorks) for further information on contract instance, behavior and state, and so on.

**Architecture overview**

With blockchain technology, whenever a new transaction is added, a consensus process occurs before the transaction is committed to the ledger. If the transaction data is large, latency increases as the consensus service does its job. This means it may take a little time for the transaction to become visible as "committed" to the monitoring systems. You may consider this delay as a consensus latency.

In addition, an application that interacts with the blockchain will typically need to be aware of which data need to be non-repudiable, and which data do not. For the data that do not need to be non-repudiable, it may make sense to store the data directly in the repositories rather than in the blockchain.

Because this effectively places a limit on the speed and volume at which data can be added to the blockchain, the real-time data from the vehicle IoT and V2V sensors will not be sent directly to the blockchain.

In the design of this demo application, we are focused primarily on blockchain processing. The real-time data coming from V2V and IoT are analyzed to identify specific events, which are then sent to the blockchain service. This ensures an appropriate separation between the real-time processing and the blockchain processing.

This demo does not include any components that require immediate real-time data access.

Figure 1 illustrates the high-level architecture of the demo application:
Sensors and the Watson IoT platform

Onboard **vehicle sensors** are responsible for monitoring car data during driving, and then sending that data to the Watson IoT platform. There are mainly two types of sensor data to be used for this demo application:

- On-board vehicle monitoring devices monitor speed, location, direction, and other required parameters for analyzing driving patterns.
- V2V and V2I sensors will receive events received from nearby vehicles or infrastructure and transmit them to the Watson IoT platform for further analysis.

We used a device simulator to simulate the sensor data required for this demo.

The **Watson IoT platform** receives all the vehicle events coming from the sensors. Bluemix-based services analyze these events to find driving patterns, and then forward the output events to an IBM Blockchain service instance for subsequent actions.

Smart contracts and IBM Blockchain

The **smart contract** used in this demo is derived from the Generic IoT Sample Contract and adapted to a car driving scenario. This smart contract runs in a blockchain based on Hyperledger Fabric and is designed for integration with the Watson IoT Platform.

As in the Generic IoT Sample Contract, two classes of message are handled in the blockchain ledger:

- **Transactions**, which change the ledger state
- **Queries**, which read the ledger state

This smart contract inherits very useful features from the Generic IoT Sample Contract, such as asset history storage, most recent update first, a rules engine to raise or clear alerts, an alerts mechanism to track the active alerts, and a “partial state as an event” pattern.
The Node.js-based **client application** provides stakeholders with access to all the data stored in the IBM Blockchain secured store as a result of executing the chaincode. Stakeholders (Insurer, Driver, Government authority) can view the output of the transactions as IoT data are received and processed by the chaincode.

The client application uses the IBM Blockchain REST API to query the chaincode and display the data through a simple web-based interface.

**Implementation details**

This section describes a detailed approach to implementing the use case mentioned above. We describe the required implementation components and how they interact to achieve the required functionality. Then we list the individual steps of the functional flow and how those steps are implemented using Bluemix services and the IBM Watson IoT Platform.

**Run the app**
*Get the client application code sample*

**Get the smart contract code sample**

Figure 2 shows the different components of the demo application.

**Figure 2. Application components**

The following components play an important role in analyzing the data coming from the sensors:

- **Node-RED orchestration**—The Node-RED Starter service in Bluemix is used to orchestrate the IoT events and send those events, after augmentation, to the driving pattern analysis service.
- **Driver Behavior analysis**—Watson IoT Driver Behavior services take the augmented event data and analyze Driver Behavior and vehicle usage patterns.

Figure 3 shows how the components interact to implement the required functionality using the Node-RED and Driver Behavior analysis services. In this scenario, we have triggered the smart contracts directly from the Node.js-based custom application.
Here’s how the process works:

1. The owner registers his or her vehicle using the client application, which in turn creates the assets in the blockchain.
2. Sensors in the registered vehicle send car probe data to the Watson IoT platform.
3. Sensor data is analyzed using Node-RED-based orchestration (details provided below), which invokes the Driver Behavior service.
4. The output of the Driver Behavior service is then used to trigger the blockchain service, which in turn executes smart contracts and performs the actions described above.

On-board IoT and V2V data collection

The V2V and IoT data will be collected by the device on board the vehicle, and the device will send the data to the Watson IoT Platform. For the demo application, we will use a device simulator to send the device driving data as shown in the code listing below.

Data from car sensors

Car sensor data is an ordered sequence of car events. Each event consists of the heading, latitude, longitude, speed, and timestamp that are retrieved from the car for a trip. Sample sensor data used for analysis of the driving pattern is shown here:

Listing 1. Sample sensor data

```json
{
    "heading":273.995431,
    "latitude":30.398981,
    "longitude":97.746804,
    "speed":8.256416399999999,
    "timestamp":"2016-03-11T13:43:17.000Z",
    "trip_id":"1HGFM82644B006001_2017-02-10"
}
```

Data from V2V sensors

Two types of simulated vehicular communication messages are sent to Watson IoT platform:

1. **Basic safety message (BSM)**—This includes V2V messages received by the receiver in the connected car. BSMs include emergency brake lights, vehicle speed, and direction.
2. **Roadside alert message (RSA)**—These messages are triggered by roadside infrastructure components. RSA messages include traffic signals, stationary vehicle warnings, incidents, and diversions.

A sample V2V BSM message is shown here:

### Listing 2. Sample V2V BSM message

```json
{
    "msgID": "basicSafetyMessage",
    "secMark":10,
    "id": 1111111111,
    "lat":60369542144,
    "lng": -984987654,
    "elev": 13000,
    "speed": 10000,
    "heading": 1820,
    "accelset": {
        "lat":0,
        "lng":0,
        "vert": 0,
        "yaw": 0
    },
    "brakes": {
        "wheelBrakes": 11110000,
        "traction": "off",
        "abs": "off"
    },
    "steering": 1501,
    "throttle": 60,
    "lightSet": {},
    "size": {
        "width": 213,
        "length": 640
    },
    "valueCnt1": 0,
    "items1": {},
    "valueCnt2": 0,
    "items2": {}
}
```

These messages are analyzed to produce driving-specific events (such as traffic signal violations, speed limit violations, and near collisions), which are passed to the [Bluemix Blockchain](https://ibm.com/developerWorks) instance to analyze and record.

### Linking V2V and IoT events with distributed ledger

Once the IoT platform receives the data, the corresponding Node-RED flow is triggered. The flow filters the data, augments it, and applies specific rules as necessary. In our demo application, the on-board data is sent by the Node-RED flow to the Driver Behavior service. The Driver Behavior service identifies the events based on trips, as a batch job. On job completion, the demo application triggers a transaction at the blockchain service. Thus in our demo application, each trip causes a transaction to be added to the blockchain.

### Node-RED workflow

Figure 4 shows the Node-RED workflow we are using in our demo application.
The purpose of this Node-RED flow is to allow IoT data to flow into the Driver Behavior service, with geographical map augmentations as necessary.

The flow reads the incoming IoT message and starts by adding the appropriate tenant ID and hostname for the Driver Behavior service that’s associated with your Bluemix application. The respective device data is extracted from the incoming IoT message and put into a format that can be understood by the Driver Behavior service.

This data is then matched with geographical map information from the Context Mapping service. Two Context Mapping APIs are invoked: /matching and /link. The generated logs show you what data are added to the message.

For our demo, we have simulated trip data (the sequence of events shown above) to be fed to the Driver Behavior service, which generates specific driving behaviors for the trip. See the documentation for a description of the generated behavior events from the sensor data.

Finally, the /carProbe API from the Driver Behavior service is invoked with the message so that the service can add the probe data against a particular trip in its data store.

The green-colored "log" nodes are for producing debug output. The corresponding “delay” nodes ensure that the log message is fully written to the output before moving on to the next node, so that the log messages do not overwrite each other.
You can get more information on the API details in the Bluemix service documentation, and you'll find more on how to set up this flow in this tutorial.

**Executing ledger transactions**

The events—and therefore the transactions—may be triggered by events initiated by the IoT platform, or by events initiated by the participants.

Once a transaction is received by a validating peer in the blockchain service, it will initiate a consensus with other validating peers in the network. In the default Bluemix Blockchain service, there are four validating peers. Once a consensus is reached, the validating peers will commit the transaction.

Based on the transactions received and committed to the ledger, a participant may initiate a follow-on transaction. For example, when a bad driving event is received, the Government authority may initiate a transaction to reduce the driving rating.

**Smart contract schema**

Smart contracts process the data coming from the sensors and the driver behavior service. The code listing below shows the sample schema used in the demo application to process all the events in the transactions. This is based on the Trade Lane sample for Advanced IoT.

**Listing 3. Sample schema used to process all transaction events**

```json
"state": {
  "type": "object",
  "description": "A set of properties that constitute a complete asset state. Includes event properties and any other calculated properties such as compliance related alerts.",
  "properties": {
    "assetID": {
      "$ref": "#/definitions/assetID"
    },
    "vehicleID": {
      "$ref": "#/definitions/vehicleID"
    },
    "timestamp": {
      "type": "string",
      "description": "Event timestamp."
    },
    "lastKnownLocation": {
      "$ref": "#/definitions/geo"
    },
    "extension": {
      "type": "object",
      "description": "Application-managed state. Opaque to contract.",
      "properties": {
        "authorityRating": {
          "$ref": "#/definitions/rating"
        },
        "insurerRating": {
          "$ref": "#/definitions/rating"
        },
        "lastTrip": {
          "$ref": "#/definitions/trip"
        }
      }
    }
  }
}
```
Developing the client application

We have used the Bluemix Node.js SDK to develop the sample client web application that interacts with and queries the data from blockchain. The client application queries blockchain to show asset and transaction details. It also triggers the simulated data that is sent to the Watson IoT platform and then to the driver analysis service to get the driving behavior rating.

Blockchain REST API call configuration

This is the configuration used by all of the REST API calls. It establishes the peer you are connecting to, the user id and secret to authenticate the API call, and the call arguments.

Listing 4. Blockchain REST API call configuration

```javascript
var config = {
    host: "<peer id>-vp0.us.blockchain.ibm.com",
    port: "5003",
    secure_context: "<user id>",
    enroll_secret: "<enroll secret>",
    protocol: "https",
    debug_mode: true,
    chaincodeURL: "<peer id>-vp0.us.blockchain.ibm.com:5003/chincode",
    name: blk_chaincode,
    path: blk_chaincode,
    messageId: "",
    contract_version: "1.0",
    timeout: 3000,
    template: {
        "jsonrpc": "2.0",
        "method": "{{method}}",
        "params": {
            "type": 1,
            "chaincodeID": {
                "name": "mycc",
            },
            "ctorMsg": {
                "function": "{{function}}",
                "args": [],
            },
        },
    },
};
```
The sections below provide a step-by-step guide to implementing the demo application. The code snippets show how to use the IBM Blockchain REST API to register assets, update the asset data, and query the blockchain for current and historical asset data.

**1a. Register an asset**

The user starts by registering the vehicle he or she has as an asset. Figure 5 shows the asset details supported by the demo application.

**Figure 5. Demo application asset details**

**1b. Create a new asset**

Blockchain method to be used: createAsset(). Also, a Cloudant database is used as the vehicle repository for the demo. The vehicle is inserted into the database after the asset is created in blockchain.

**Listing 5. createAsset()**

```javascript
// Prepare the rest body for createAsset
createBody.method = 'invoke';
createBody.params.ctorMsg.function = 'createAsset';
createBody.params.ctorMsg.args = ['{"assetID":"'+ input_vin +"","ownerName":"'+ input_ownerName +"","createDate":"'+ v_createdt +"","numberPlate":"'+ input_numberPlate +'"}'];
```
First, call the blockchain createAsset function using a REST API:

```javascript
https.request(options, function(res4) {
  res4.setEncoding('utf8');
  res4.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});
```

Then prepare the vehicle JSON structure for inserting into Cloudant:

```javascript
var vehicle = {
  vin: input_vin,
  owner: input_ownerName,
  number: input_numberPlate,
  status: 'active',
  createDate: v_createdt
};
var v_id = input_vin + '-' + input_numberPlate;
```

Insert the vehicle into Cloudant:

```javascript
db.insert(vehicle, v_id, function(err, body, header) {
  if (err) {
    return console.log(v_id + ' -- [insert] ', err.message);
  }
});
```

This completes the preparation of the asset for tracking. Step 2 shows you how to capture and augment the IoT data.

### 2a. Send simulated vehicle data for registered vehicles

Once a vehicle is registered, you need to send the on-board vehicle sensor data. A device simulator is used to send car probe data to the Watson IoT platform. Refer to the device simulator code for simulator car driving data.

This code snippet shows how to send data to the Watson IoT platform:

**Listing 6. Sending data to the Watson IoT platform**

```javascript
deviceClient.connect();
deviceClient.on('connect', function(){
  if(iotDeviceConnectionStatus == false) {
    iotDeviceConnectionStatus = true;
    var i=0;
    console.log("connected");
    var IOTSent = setInterval(function sendIOTData() {
      if (i >= totalcount) {
        deviceClient.disconnect();
        iotDeviceConnectionStatus = false;
        selectedVehicle.sim_status_id="1";
        iotDataSent = true;
        clearInterval(IOTSent);
        return;
      }
      datatopublish.d = carprobesampledata[i];
      i++;
      processingStatus = "Capturing car data "+i="/"+totalcount;
      console.log(processingStatus);
      var driveTimestamp = new Date().toISOString();
      datatopublish.d.timestamp = drivedate + driveTimestamp.substr(10,driveTimestamp.length);
```
2b. After the raw sensor data has been received by the IoT platform, augment the same with context mapping data

The custom Node-RED flow described above listens to the IoT platform events. The flow receives the sensor data from the IoT platform. We have implemented the flow to invoke the Context Mapping service and augment the data.

2c. With augmented data, send it to the driver analysis

The Node-RED flow then sends the augmented sensor data to the Driver Analysis service to store the data.

Next, you need to coordinate the analysis of the data and update the analysis results in the blockchain asset. In this tutorial, that coordination is done by the front-end simulator demo application.

3a. When the data set is ready in Driver Analysis, it's time to trigger the analysis of the data

The user should continue to see the status updated in the UI.

The driver analysis job submission API (POST) URL is as follows:

https://automotive.internetofthings.ibmcloud.com/driverinsights/jobcontrol/job?
tenant_id=yourtenantid&from= dateoftravel&to=dateoftravel
Authorization Basic userid:secretkey

This userid and secretkey are from the Driver Behavior service that you have attached to your Bluemix application. The "to" date can also be later than the "from" date, if you want to analyze multiple days of data in a single batch.

Listing 7. "to" date and "from" date

```javascript
var new_event_date = moment(selectedVehicle.lastEventDate, 'DD-MMM-YYYY').add(1, 'days').format('YYYY-MM-DD');
path_str = "&from=" + new_event_date + "&to=" + new_event_date;
console.log(path_str);
driver_path_post = "https://automotive.internetofthings.ibmcloud.com/driverinsights/jobcontrol/job?tenant_id=<tenant_id>"+path_str;
request( 
    { 
        url : driver_path_post, 
        method : 'POST'
    })
```
headers: {
  "Authorization": auth,
  "content-Type": 'application/x-www-form-urlencoded'
},
}

function (error, response, body) {
  var jsonbody = JSON.parse(body);
  selectedVehicle.job_id = jsonbody.job_id;
  selectedVehicle.sim_status_id = "2";
  selectedVehicle.sim_status_desc = "Driver analysis in progress. This might take couple of minutes. Job id = " + jsonbody.job_id;
  res7.render('index', { title: 'Home', moment: moment, selectedVehicle: selectedVehicle, vehicleList: db_vehicle_list, sim_triggered: 'true'});
});

This returns a job_id in a JSON like this:

```
{
  "job_id": "4f9b70ef-6bcd-4a21-b311-e57ba9729a3d"
}
```

With this job_id, wait for the driver analysis to finish. This can take about a minute with our sample data.

Here is the job status API (GET) URL:


The code snippet is shown here:

**Listing 8. Job status API**

```
path_str = "&job_id=" + selectedVehicle.job_id;
driver_path_get = "https://automotive.internetofthings.ibmcloud.com/driverinsights/jobcontrol/job?tenant_id=<tenant_id>" + path_str;
request(
  {
    url: driver_path_get,
    method: 'GET',
    headers: {
      "Authorization": auth,
      "content-Type": 'application/x-www-form-urlencoded'
    }
  },
  function (error, response, body) {
    var jbody_1 = JSON.parse(body);
    var sim_stat = jbody_1["job_status"];
    console.log("job status - " + sim_stat);
    if(sim_stat="SUCCEEDED")
    {
      selectedVehicle.sim_status_id = "3";
      selectedVehicle.sim_status_desc = "Driver analysis is complete.. Fetching results...";
    }

    The returned JSON will contain the job status as "RUNNING" or "SUCCEEDED." If it is in "RUNNING" state, call the same API after an interval.
```
Once the driver analysis results are available, the resulting data structure needs to be transformed into a data structure that will be accepted by the chaincode we have deployed.

In our sample code, this structure is kept mostly the same for brevity; no major transformation is needed.

### 3b. Obtain the driver analysis results

For fetching trip details, two calls are made to driver service API. The first call is made with the tenant id and job id to retrieve the trip uuid:

```
```

And the second call is made with the trip uuid to retrieve the trip details:

```
```

The driver and insurer ratings are updated in this data structure. The rating is reduced if bad driving events are detected, and the rating is increased if no events are detected, indicating good driving.

### 3c. Place request to blockchain with the new asset data against the asset ID and asset state should get updated

Next, the `updateAsset` method of our chaincode is called, with an array of strings as the input argument. The length of the array is 1. So the zero-th element of the array will contain the data structure obtained in the above step.

#### Listing 9. Calling the `updateAsset` method

```javascript
// Prepare the rest body for updateAsset
updateBody.method = 'invoke';
updateBody.params.ctorMsg.function = 'updateAsset';
updateBody.params.ctorMsg.args = "{"assetID":""+ input_vin +"","lastEvent":""+ input_lastEvent +""}"

// Call blockchain updateAsset function using rest api
https.request(options, function(res4) {
    res4.setEncoding('utf8');
    res4.on('data', function (chunk) {
        console.log('BODY: ' + chunk);
    });
}).end(JSON.stringify(updateBody));
```

### 4. Query the blockchain to get the asset state

Call `readAsset()` with the asset id to obtain the latest ratings, and show it on the UI.

#### Listing 10. `readAsset()`

```javascript
var queryBody = config.template;
```
queryBody.params.chaincodeID.name = blk_chaincode;
// Prepare the rest body for readAsset
queryBody.method = 'query';
queryBody.params.ctorMsg.function = 'readAsset';
queryBody.params.ctorMsg.args = ["{"assetID":"\"+selected_vin+\"\"}""];

console.log('queryBody - ' + JSON.stringify(queryBody));
var raw_str;
// Call blockchain readAsset function using rest api
https.request(options, function(res2) {
    res2.on('data', function (chunk) {
        console.log('BODY: ' + chunk);
        raw_str+=chunk;
        var json_str = eval('(' + chunk + ')');
        var json_msg = eval('(' + json_str.result.message + ')');
        selectedVehicle.assetID = json_msg.assetID;
        selectedVehicle.createDate = json_msg.createDate;
        selectedVehicle.ownerName = json_msg.ownerName;
        selectedVehicle.numberPlate = json_msg.numberPlate;
        selectedVehicle.lastEventDate = moment(new Date(json_msg.txntimestamp)).format('DD-MMM-YYYY');
    });
}).end(JSON.stringify(queryBody));

### Query the blockchain to get the asset state history

As shown in the following code snippet, the history data is retrieved by invoking the `readAssetHistory()` method of the chaincode. Using the history data, one can list the past trips against the vehicle, and can show the trip on a map as per the latitudes and longitudes.

**Listing 11. Retrieving the history data**

```javascript
queryBody.method = 'query';
queryBody.params.ctorMsg.function = 'readAssetHistory';
queryBody.params.ctorMsg args = ["{"assetID":"\"+selectedVehicle.assetID+\"\"}""];

// Call blockchain readAsset function using rest api
https.request(options, function(res2) {
    res2.on('data', function (chunk) {
        console.log('BODY: ' + chunk);
        var jbody_1 = JSON.parse(chunk);
        var json_msg = jbody_1[0].result.message;
    });
}).end(JSON.stringify(queryBody));
```

### Conclusion

As demonstrated in this tutorial (and the code samples), it is relatively easy to quickly build a cognitive IoT application that integrates with an IBM Blockchain service instance. Data from IoT sensors can be analyzed and fed into blockchain smart contracts for running rules and complete transactions that can be validated and agreed on by all the participating peers.

This application can be further enhanced by including additional cognitive services that integrate, for example, weather data during a trip or driving restrictions at different locations, to further analyze how drivers respond under different conditions. Similar solutions can also be used to analyze the performance of new drivers in training scenarios.
Related topics

- Part 1 of this series
- Blockchain Developer Center
- Blockchain service on Bluemix
- Get the client application code sample
- Get the smart contract code sample
- Basic smart contract sample
- Developing smart contracts for the Watson IoT Platform
- Hyperledger Fabric Client (HFC) SDK for Node.js
- IoT Context mapping service in Bluemix
- Driving Behavior Analysis service in Bluemix

© Copyright IBM Corporation 2017
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)