Javascript Smart Contracts

This document introduces some general notions to Kalima smart contracts, and explains how to create Javascript smart contracts for Kalima. Noted that Javascript smart contracts are launched by Java client nodes, and that this document does not show how to create your own node to launch smart contracts.

However, a node has been installed and allows you to launch smart contracts. One is launched when new transactions arrive at /sensors, the other is launched when new transactions arrive at /alarms/fire.

1. Prerequisite

To create a Smart Contract it is necessary to have a code editor and / or a text editor:

- Example of a code editor: Visual Studio Code, download link: https://code.visualstudio.com/
- Example of a text editor: notepad++, Gedit (Ubuntu),vi, nano...

It is also necessary to install git, since contracts are stored on git directories. To modify a contract, simply push the contract to git, then wait a few minutes, while the contract is deployed, as described in the documentation API_Kalima.

2. Creating a JavaScript Smart Contract

Initialisation:

To ensure that your JavaScript Smart Contract works, you need to add the following line to the beginning of your program:

```
load("nashorn:mozilla_compat.js");
```

Import packages:

Javascript smart contracts are launched by java nodes. From smart contracts, we have access to the Kalima Java API. In addition, one can import any Java package.

To import a Java package you can use the "importPackage()" function as shown in the following example:

```
importPackage(Packages.java.io);
importPackage(Packages.java.lang);
importPackage(Packages.java.util);
importPackage(org.kalima.kalimamq.message);
importPackage(org.kalima.cache.lib);
importPackage(org.kalima.util);
```

Creating Java Objects:

To follow up on importing Java packages, you will need to declare the objects Java to use in your Smart Contract in the same way as the following example:

```
var JString = Java.type("java.lang.String");
var KMsg = Java.type("org.kalima.cache.lib.KMsg");
```

The smart contract code:

You can then write the code for your smart contract. Noted that the java node that will be responsible for executing the smart contract, will necessarily perform a function of the smart contract. Your code must therefore be inside a function.

3. Example of Smart Contract

As part of the tutorials, there are as said at the beginning of this document two smart contracts. These smart contracts are ona private git server, in the repository "KalimaContractsTuto". If you do not have access to this directory, please contact a member of our team. If you have access, you can modify existing smart contracts to change their behaviors. For your changes to be taken into account, you will only have to commit the changes on the repository and wait a few minutes for the process of publishing smart contracts to be completed.

We also find the two examples of smart contracts on GitHub: https://github.com/Kalima-Systems/Kalima-Tuto/tree/master/KalimaSmartContracts.

The "sensor.js Smart Contract":

Here is the body of the sensors.js example:

```
function main(kMsg, clone, logger) {

	if(!kMsg.getKey().equals("temperature2")) return;

	var body = new JString(kMsg.getBody());

	var temperature = parseInt(body, 10);

	if(temperature == null)

		return "NOK";

	if(temperature >= 100) {

		clone.put("/alarms/fire", kMsg.getKey(), ("Temperature too high: " + temperature + " °C").getBytes());

	}

}
```

It is executed with each new transaction on the /sensors address. Three parameters have passed to it:

- kMsg: The message corresponds to the transaction received. In particular, it will make it possible to recover useful data, or the key to the transaction
- clone: The clone of the node that launches the smart contract. It allows to read the transactions of the blockchain and to create new transactions for example
- logger: Kalima's internal logger. It is not useful for this tutorial because you do not have access to the smart contract node. However, when you deploy your own node, it allows you to add logs to log files.

The contract is simple: We start by checking the key of the message. If it is not equal to "temperature2", nothing is done. Then we retrieve the useful data of the transaction on the assumption that it corresponds to an integer that corresponds to a temperature for example. Finally, a new transaction is created at the address "/alarms/fire" if the temperature exceeds a certain threshold.

For example, as part of the tutorial, you can change the temperature threshold and then send temperatures using the java example to verify that the threshold has actually changed.

The "fire.js Smart Contract":

The smart contract fire.js does nothing at the moment, except record in the logs the content of the alarms. You are free to modify this smart contract as a training.