Exemple Java

1. Prerequisite

To test the Java example, it is necessary to have some prerequisites:

- An integrated development environment such as Eclipse. Download link: https://www.eclipse.org/downloads/packages/
- A Java Development Kit JDK. It is advised to have at least the version 11 of the jdk to run the example successfully

To start, check that the Kalima.jar is included in your project (in the /libs folder). Then, right click on the jar ➔ Build Path ➔ Add to Build Path.

You have now access to the Kalima API in your project.

2. SerialId

The serialId is an identifier whose use will be explained in another paragraph of this document.

Note that to connect to Kalima blockchain it is necessary to use your own serialId communicated in the free trial email (insert it in the configuration file .config).

3. Execution of the KalimaJavaExample project

To test the project downloaded from github "KalimaJavaExample", you can follow the video accessible via this link : https://www.youtube.com/watch?v=1pyACOm1lTI ;

Or follow these steps :

- Start Eclipse,
- Click on the File menu ➔ Import ➔ General ➔ Existing Project sinto Workspace ➔ Next ➔ Click on Browse in the Select root directory field ➔ Click on the KalimaJavaExample repository downloaded from github ➔ Press on Select Folder ➔ Finish,
- From Eclipse open the file KalimaJavaExample/etc/cfg/node.config via a text editor -> Change the SerialId with the one you received in the free trial email,
- From Eclipse open the file KalimaJavaExample/src/Client.Java,
- Click on the Run menu ➔ Run Configurations ➔ Right click on JavaApplication ➔ New Configuration ➔ Open the Arguments tab ➔ Indicate the relative path (or the complete path) of the config.node file (etc/cfg/node.config) in the Program arguments field ➔ Click on Run,

When you run the project you will be asked if you want to use Smart Conrtats:

- If you choose "Y" : you need to enter your git ID sent in the free trial email (using your new git password),
- If you choose "N", you will not need to enter anything.

To do the command line execute of the you can go to the section " Code execution ; Command line execution".

The execution result of the KalimaJavaExample project can be found in the "Results" section of this document.

4. Code explanation

a. Initialisations

The code below allows you to initialize a certain number of objects and to connect to the Blockchain.

```
public Client(String[] args) {
	clonePreferences = new ClonePreferences(args[0]);
	logger = clonePreferences.getLoadConfig().getLogger();
}


public void initComponents(String[] args){
	node = new Node(clonePreferences.getLoadConfig());
	clone = new Clone(clonePreferences, node);

	clientCallBack = new KalimaClientCallBack(this, gitUser, gitPassword);

	Properties prop = new Properties();
	String propFileName = args[0];
	InputStream inputStream;
	try {
		inputStream = new FileInputStream(propFileName);
		prop.load(inputStream);
	} catch (FileNotFoundException e) {

	} catch (IOException e) {

	}
	this.ScriptPath = prop.getProperty("SCRIPT_PATH");

	try {
		node.connect(null, clientCallBack);
	} catch (IOException e) {
		logger.log_srvMsg("ExampleClientNode", "Client", Logger.ERR,
"initComponents initNode failed : " + e.getMessage());
	}
}
```

The Node will be responsible for the connexion with the Blockchain

The clone is responsible for the synchronisation of the data in cache memory.

The clientCallBack allows you to react to the addition of new transactions in the Blockchain (see CallBacks chapter).

We can see that we have to pass args[0] during the creation of the clonePreferences. Indeed, you must launch your client by passing the path of a configuration file. We will talk later about the content of this file.

Finally, the properties will allow us to retrieve information from the config file. We'll see more details later when explaining the config file.

-

b. Callbacks

As we have seen previously, we have to pass two callback classes to the Node object. The serverCallback is not useful for an ordinary node. So you can just create a simple class that inherits ServerCallback, without putting anything in the methods.

The ClientCallBack is more important and requires a few necessary lines. It will allow you to react to the arrival of new transactions. In the code here, we have a class that inherits from ClientCallback. The functions that we are not interested in are left empty. The functions that interest us for this example are putData, onConnectionChanged and onNewCache.

The putData function will be called at each new transaction received, you must at least add the code below, and then customize the code according to your needs.

```
KMsg kMsg = KMsg.setMessage(msg);
client.clone.set(kMsg.getCachePath(), kMsg, true, false);
```

The onConnectionChanged function will be called at every connection / disconnection with one of the Notary Nodes. You must at least insert the code below, and you can add more if needed.

```
client.clone.onConnectedChange( (status==Node.CLIENT_STATUS_CONNECTED) ? new
AtomicBoolean(true) : new AtomicBoolean(false), nioClient, false);
```

The onNewCache function is called every time a new Cache is created in our Node. All caches will be created at the beginning of the connection, during synchronization. We can create callbacks for each Cache. In this example, a callback has been created to manage Smart Contracts. We subscribe to this callback in the onNewCache function :

```
client.getClone().addListnerForUpdate(new
SmartContractCallback(cachePath, client, contractManager));
```

c. Smart Contracts (SmartContractCallback)

Smarts contracts are stored on git but validated by the Kalima blockchain. All the management of these smarts contracts is integrated in the Kalima API. To be able to execute smarts contracts from our Node, you just have to provide the login information (login, password) of an authorized account on the git directory where the smarts contracts are stored.

To use the Smart Contracts you are going to create on the Kalima blockchain, you have to share them on Git in your proper directory on which you will be authorized.

In the example, the identifiers are requested at the start of the application and we fill in this information with the code below.

```
contractManager.loadContract(GIT_URL, GIT_USERNAME, password,
kMsg.getKey(), kMsg.getBody());
```

Once loaded, a smart contract can be executed :

```
private void runContract(KMsg kMsg) {
	if(client.getNode().getSyncingKCaches().get(kMsg.getCachePath()))
		return;
	String scriptPath = logger.getBasePath() + "/git/KalimaContractsTuto" +
kMsg.getCachePath() + ".js";
	try {
		String result = (String) contractManager.runFunction(scriptPath,
"main", logger, kMsg, client.getClone(), client.getNode());
		logger.log_srvMsg("ExampleClientNode", "TableCallback",
Logger.INFO, "script result=" + result);
		System.out.println("script " + scriptPath + " result=" + result);
	} catch (Exception e) {
		logger.log_srvMsg("ExampleClientNode", "TableCallback",
Logger.ERR, e);
	}
}
```

In our example, the runContract function is called every time a new data arrives in the Node. The node will launch the script having the name of the cache path in which the data was received, if it exists. For example, if we receive a data in /alarms/fire, the node will launch the KalimaContractsTuto/alarms/fire.js script.

Bindings are used to pass objects to the scripts. In this example, we pass a KMsg (the received message), a Logger, the clone and the node.

To have more details about Smart Contracts you can consult the documentation shared in the following path : https://github.com/Kalima-Systems/Kalima-Tuto/blob/master/etc/doc/fr/Les_Smart_Contracts.pdf

5. The configuration file

Here is an example of configuration file :

```
LedgerName=KalimaLedger
NODE_NAME=Node Exemple

NotariesList=167.86.124.188:9090,62.171.130.233:9090,62.171.131.157:9090,144.91
.108.243:9090
FILES_PATH=/home/rcs/jit/KalimaExample
SCRIPT_PATH=/sensors
SerialId=PCTuto
#WATCHDOG=600000
SEND_TIMEOUT=10000
```

- LedgerName ➔ Is not yet used in the current version
- NODE_NAME ➔ You can put something that allows you to recognize your node
- NotariesList ➔ Comma-separated list of notary's addresses and ports

- FILES_PATH ➔ This is the path where the files useful to Kalima will be stored, as well as the logs
-
- SCRIPT_ PATH ➔ This is the path of the smart contract in the private git. For example here, the executed smart contract will be <gitname>/sensors.js
- serialId ➔ It is an identifier that will allow the authorization on the blockchain at the first launch of the client node (provided by Kalima Systems in the case of a test on our Notary).
- WATCHDOG ➔ The watchdog here is commented out, it is not useful for our example. The watchdog is useful in case of prolonged connection with the Blockchain. It corresponds to a time (here, 600000 = 600 seconds = 10 minutes). In fact, every 10 minutes, we will check that the connection is still active. You can then for example make a Smart Contract that will send you an email if the connection is lost. It is thus useful to check the state of the network.
- SEND_TIMEOUT ➔ Corresponds to the maximum time considered for sending a message. If at the end of the time indicated, the message is not well received, we try to send it again (here 10 seconds).

6. Code execution

To test your project, you can run the code from Eclipse, or from a command line console. You just have to pass as a parameter, the path of the configuration file.

a. Execution from Eclipse

In Run ➔ Run Configurations ➔ right click on « Java Application » ➔ New Configuration.

- ⇨ Choose a name for the configuration
- ⇨ under « Project » clic on « Browse » and choose your project
- ⇨ under « Main class » clic on « Search » and select the class including the Main method that you will launch (here : Client.java)
- ⇨ In the "Arguments" tab, under "Program arguments", give the path of the config file (here: etc/cfg/node.config)

Probably the only thing you have to change is the argument. Everything else should already be done at runtime. If this is not the case, follow all the instructions.

Your configuration is ready, you can execute it.

b. Command line execution

You can also generate the jar and then execute it on the command line. On Eclipse, right click on your project ➔ Export, choose Java ➔ Runnable Jar File ➔ Next.

In the "Runnable JAR File Export" window, choose your configuration under "Launch Configuration", and choose a destination for your jar (ex: /Documents/git/KalimaTuto/TutoClient/etc/jar/TutoClient.jar), then click on "Finish".

then, from the console:

```
cd /Documents/git/KalimaTuto/TutoClient/etc/
java -jar jar/TutoClient.jar cfg/node.config
```

7. Results

The example program connects to the Blockchain and sends 10 messages (1/second). The TTL (Time To Live) of these messages is 10, which means that each message will be automatically deleted after

```
Do you want use Smart Contracts ? (Y/n)
```

10 seconds (a transaction will take place on the blockchain for each deletion). So, if your code is correct, you have correctly configured the configuration file, and your device is authorized on the blockchain, you should have something similar in your console after 2 seconds :

If you choose « n », you will have :

```
putData cachePath=/alarms/fire key=BEGINMEMCACHE body=/alarms/fire
[…]
putData cachePath=/Kalima_Password_json key=ENDMEMCACHE
body=/Kalima_Password_json
GO
putData cachePath=/sensors key=key0 body=95
putData cachePath=/sensors key=key1 body=96
putData cachePath=/sensors key=key2 body=97
putData cachePath=/sensors key=key3 body=98
putData cachePath=/sensors key=key4 body=99
putData cachePath=/sensors key=key5 body=100
putData cachePath=/sensors key=key6 body=101
putData cachePath=/sensors key=key7 body=102
putData cachePath=/sensors key=key8 body=103
putData cachePath=/sensors key=key9 body=104
putData cachePath=/sensors key=key0 body=
putData cachePath=/sensors key=key1 body=
putData cachePath=/sensors key=key2 body=
putData cachePath=/sensors key=key3 body=
putData cachePath=/sensors key=key4 body=
putData cachePath=/sensors key=key5 body=
putData cachePath=/sensors key=key6 body=
putData cachePath=/sensors key=key7 body=
putData cachePath=/sensors key=key8 body=
putData cachePath=/sensors key=key9 body=
```

If you choose « Y » you will have :

```
putData cachePath=/alarms/fire key=BEGINMEMCACHE body=/alarms/fire
[…]
putData cachePath=/Kalima_Password_json key=ENDMEMCACHE
body=/Kalima_Password_json
GO
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key0 body=95
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key1 body=96
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key2 body=97
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key3 body=98
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key4 body=99
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key5 body=100
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key6 body=101
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key7 body=102
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key8 body=103
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key9 body=104
putData cachePath=/sensors key=key0 body=
putData cachePath=/sensors key=key1 body=
putData cachePath=/sensors key=key2 body=
putData cachePath=/sensors key=key3 body=
putData cachePath=/sensors key=key4 body=
putData cachePath=/sensors key=key5 body=
putData cachePath=/sensors key=key6 body=
putData cachePath=/sensors key=key7 body=
putData cachePath=/sensors key=key8 body=
putData cachePath=/sensors key=key9 body=
```

The program starts really once "Go" is displayed.

In our example the client sends 10 messages in 10 seconds. The messages will be received by all authorized nodes on the path cache in question, including yours. Thus, you must see in the logs a line for each message sent (lines starting with "StoreLocal"). For each message received in /sensors, we launch the sensors.js script. If you answered "Y" at the beginning of the program, and if you have correctly identified on git, you should see in the logs the results of the script.

Finally, the messages will be deleted one by one, since the TTL has been set to 10 seconds. So you must see the transactions in the logs (lines starting with "StoreLocal remove").

If nothing happens after « Go » there is several possibilities :

- You are not authorized on the blockchain
- You made a mistake on the config file (make sure to have a valid serialId)
- You are not connected to internet