Develop a smart contract node

Prerequisite

To use the Kalima Java API, it is recommended that you have previously read the documentation API_Kalima.

To develop a smart contract node, it is better to read the documentation API_Java, because the Java API can be used in smart contracts to create new transactions for example.

A complete example of a smart contract node is available on our public GitHub: SmartContractNode

Configuration

The Kalima api is provided in the form of a JAR, there are two versions:

- Kalima.jar Contains the necessary elements to create a Kalima node capable of connecting to a Kalima Blockchain, performing transactions on the blockchain, and reacting in an event-driven way to the creation of new transactions (this makes it possible to create Java smart contracts) ➔
- KalimaSC.jar Also includes the ContractManager which offers the possibility to launch smart Javascript or python contracts

To use the Kalima api in your project, simply include the jar of your choice in your dependencies.

For this example, we will take KalimaSC.jar.

Contract Manager

The KalimaSC API, unlike the Kalima API, embeds a ContractManager module.

This module allows you to detect the arrival or modification of new contracts, then decrypts, verifies, and installs them. It also makes it possible to launch smart contracts according to the rules of our choice.

If you used our tutorial inscription form (https://inscription.tuto.kalimadb.com/airdrop), you received a list of 5 addresses in the blockchain, for example: /username/addr1. To recreate this tutorial at home, you just need to replace “username” to match your addresses.

For this example, we will create a smart contract node that will execute a Javascript smart contract when creating new transactions on the blockchain in the /username/addr1 address, and a python smart contract when creating new transactions in the /username/addr3 address. Three parameters will be to our contract:

- The message received in the form of a KMsg, which will process the data received in the smart contract
- The clone of the node, which will allow for example to create transactions, or to read other data from the blockchain
- A logger, allowing you to add logs to log files

Implementation

We will detail here two types of implementations: a basic implementation, that is to say with the generic code provided, and an implementation "from scratch" to allow in-depth development to better meet your technical needs.

Basic implementation

You will find a basic example ready to use, which you can directly import into Eclipse for example:

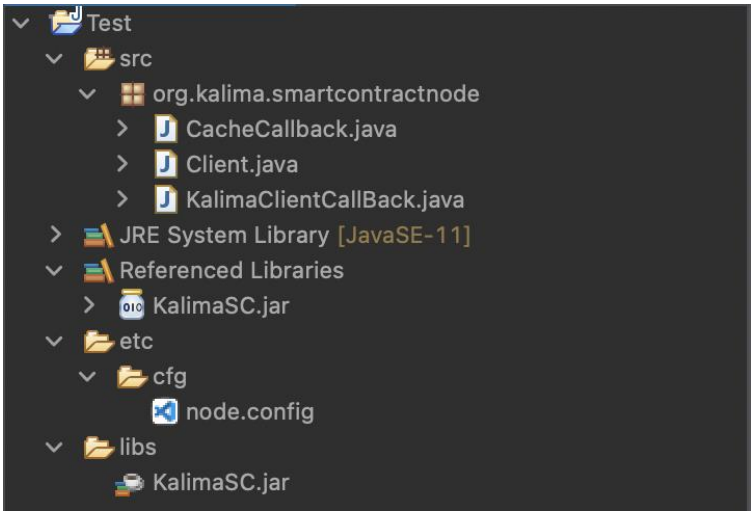
The code consists of three classes: a Client class, a CacheCallback class, and a KalimaClientCallBack class.

Launching the node

You need to make a local copy of the code, then launch it on your editor (Warning: this project has been tested since Eclipse).

When launching Eclipse, click on the "File" tab and then "Import" to open your local project.

It breaks down as is:



It is a tree structure of a classic Java project, by adding a node.config file for the configuration of the node.

A basic file is provided. This must be modified according to your testing needs before you can launch the code. For more information about this file and how to configure it, see the "Configuration File" section below.

Then, there is a variable USERNAME in Client.java that you need to modify to match your username. For example, if you are authorized on /Kalima_Contracts/John address, set USERNAME to John. Then you need to push example Smart Contracts in the git repository that you can find in the mail too.

Once the node is set correctly, you can launch the main method with the green arrow "Run" by taking care to insert the arguments: If you run without modification, etc/cfg/node.config (the path of the configuration file). Click on the drop-down menu next to the "Run" button, then on "Run Configurations", and finally on the "Arguments" tab to provide the arguments to the main method. We can then launch the program.

Configuration file

To initialize a Kalima node, you must provide it with some information that is loaded from a configuration file, for example:

https://github.com/Kalima-Systems/Kalima-Tuto/blob/master/SmartContractNode

```
SERVER_PORT=9100
FILES_PATH=/home/rcs/jit/SmartContractNode
# CHANGE IT
SerialId=YouSerialID
PRIVACHAIN=org.kalima.tuto
```

- SERVER_PORT: Each Kalima node is composed of several "clients", and a "server". Although in the majority of cases, the server part will not be used for a Kalima client node, a port for the server must be specified. You can choose the port you want, taking care to choose a port that is not already used on the machine.

- FILES_PATH: Specifies the folder in which the files necessary for the operation of the node will be stored. In particular, you will find the logs of the application.

- SerialId: For the connection to succeed, your node must be authorized on the blockchain. To initiate the connection, a Kalima administrator must create a temporary authorization (valid for 5 minutes). This temporary authorization is done through the SerialId. One can allow a node on a list of addresses, read or write. The node will therefore have access to transactions from all addresses on

which it is allowed to read or write, but it will be able to create new transactions only on the addresses on which it is authorized to write.

- PRIVACHAIN: The name of the privachain on which you want to connect your node. For tutorials: org.kalima.tuto

Implementation "from scratch"

It is considered here that the Kalima node is already initialized. Otherwise, you must refer to the documentation API_Java to correctly configure your node.

MemCacheCallback

You can start by creating a class that implements the MemCacheCallback interface. An instance of this class will then be created for each address on which our node is allowed.

This class will allow us to react to the arrival of new transactions. It is therefore here that we will decide to execute our contracts. To react to the arrival of new transactions, we will implement our code in the putData function which takes 2 parameters:

- key of type String The message key ➔
- msg of type KMessage➔ The message received

The runFunction function of the ContractManager object allows to execute a smart contract, it takes two mandatory parameters:

- The name of the contract as a String.
- The name of the function you want to launch inside the contract. In our case, we will always launch the "main" function.

You can then add as many parameters as you want. These parameters will be passed to the function that one wishes to perform in the contract. In our case, we will pass the received message, the clone and a logger, as explained above.

Smarts contracts can be pushed in many git repository, and the information about these contracts can be stored in many blockchain addresses, starts with /Kalima_Contracts (ex: /Kalima_Contracts/Kalima). this information allow the contract manager to dowload , verify and decrypt contracts.

This will give:

```
if(kMsg.getAddress().equals(client.getContractCache())) {
	client.getClientCallBack().getContractManager().downloadContract(kMsg.getProps().getProps());
}	client.getClientCallBack().getContractManager().runFunction("addr1.js", "main", kMsg, client.getClone(), logger);
} else if(kMsg.getAddress().equals("/" + Client.USERNAME + "/addr3")) {
	client.getClientCallBack().getContractManager().runFunction("addr3.py", "main", kMsg, client.getClone(), logger);
}
```

You can find the full example of this implementation here:

https://github.com/Kalima-Systems/Kalima-Tuto/blob/master/SmartContractNode/src/org/kalima/smartcontractnode/CacheCallback.java

ClientCallback

Then we can create a class that implements the ClientCallback interface. Two functions will interest us in this interface: onNewCache and onCacheSynchronized.

At startup, the node synchronizes with the master nodes. That is, it will receive the data on the addresses to which it is authorized.

When data arrives at a new address, the onNewCache function is called. We will be able to create our CacheCallback instances in this function:

```
client. getClone().addMemCacheCallback(new CacheCallback(cachePath, client));
```

When a cache is synchronized, that is, when all the data from an address has been received, the onCacheSynchronized function is called. In addition, contract information is stored at /Kalima_Contracts/... . So, we will use this function to initialize our ContractManager when we have all the information related to the contracts. And we also download all contracts here:

```
if(cachePath. equals(client.getContractCache)) {
      contractManagerRun = true;
      contractManager = new ContractManager(logger, , logger.getBasePath()
new ContractCallback() {

            @Override
            public Properties getContractInfos(String key) {
                  KMsg contractInfosMsg = client. getClone().
get("/Kalima_Scripts", key);
                  if(contractInfosMsg == null) {
                        System. out. println("contract infos not found for
" + key);
                        return null;
                  }

                  return contractInfosMsg. getProps(). getProps();
            }
      });
      for(KMessage msg :
client.getClone().getMemCache(address).getKvmap().values()) {
      client.getClientCallBack().getContractManager().downloadContract(KMsg
.setMessage(msg).getProps().getProps());
      }

}
```

You can find the full example of this implementation here:

https://github.com/Kalima-Systems/Kalima-Tuto/blob/master/SmartContractNode/src/org/kalima/smartcontractnode/KalimaClientCallBack.java

Client

Finally, we can create our main class, which we will call here "Client" and which will initialize the node:

https://github.com/Kalima-Systems/Kalima-Tuto/blob/master/SmartContractNode/src/org/kalima/smartcontractnode/Client.java

Now your Java Node is complete, but it calls 2 smart contracts that don't exist (addr1.js and addr3.py). In the next part, we explain how you can develop Javascript and Python Smart Contracts. For Python smart contracts, it's mandatory to install GraalVM and run your node with the Java SDK includes with GraalVM.

Develop contracts

Javascript

This is an example of Javascript smart contract that create a transaction in /username/addr2 when the received temperature is above 75:

```
var JavaString = Java.type("java.lang.String");

function main(kMsg, clone, logger) {

	var body = new JavaString(kMsg.getBody());
	var temperature = parseInt(body, 10);
	if(temperature == null)
		return "NOK";

	if(temperature >= 75) {
		clone.put("/username/addr2", kMsg.getKey(), "High temperature");
	}
}

({
	main: main
})
```

In Javascript smart contracts, you can use Java objects, see the first line where we used the String object of Java.

As we see before, our Java node will run functions in our smart contract. Si you need to set these functions in your script (here: main), and export them to be accessible from Java, like this: ({ id : functionName, id2 : functionName2, ...)}.

You can all variables you want for your function. Here we pass the received transaction so we can analyze its content and verify the temperature for example, we pass the clone object also so we can create and read transactions for example and we pass a logger if we want to create new logs.

Python

Here the same script but with Python:

```
import java
JavaString = java.type('java.lang.String')

def main(_, kMsg, clone, logger):
	body = JavaString(kMsg.getBody())
	print(body)
	temperature = int(body)
	if temperature is None:
		return "NOK"

	if temperature >= 75:
		clone.put("/username/addr4", kMsg.getKey(), "High temperature")
		return “OK”

type('obj', (object,), {
	'main': main
})()
```

Again, you can use Java objects as we can see in the two first lines.

You also need to set your functions and export them, but the syntax is different.

A useless parameter is added as the first parameter of your function, you can use _ to name it.

Deploy

To deploy your contracts, you need to push them into the git repository that we send you by email. Then, a DevOps tool will push your contract into the blockchain. This action can take several minutes.

Run contracts

To run Javascript contracts you just have to run your java program normally, but for Python contracts you need to install GraalVM (currently working with GraalVM 22.3.1, there is an issue with the last version): https://github.com/graalvm/graalvm-ce-builds/releases

From Linux :

```
wget https://github.com/graalvm/graalvm-ce-builds/releases/download/vm-22.3.1/graalvm-ce-java11-linux-amd64-22.3.1.tar.gz

tar -xzvf graalvm-ce-java11-linux-amd64-22.3.1.tar.gz
```

Then export JAVA_HOME and PATH env variables (see https://www.graalvm.org/latest/docs/getting-started/linux/)

Then:

```
gu install python
```

Now, run your node with the Java SDK includes with Graal, if you have exported the JAVA_HOME env variable:

```
Java -jar SmartContracNode.jar etc/cfg/node.config
```