Développer un smart contract node

Prérequis

Pour utiliser l'API Java Kalima, il est recommandé d'avoir préalablement lu la documentation API_Kalima.

Pour développer un smart contract node, il est préférable d'avoir lu la documentation API_Java, car l'API Java pourra être utilisée dans les smart contrats pour créer de nouvelles transactions par exemple.

Un exemple complet de smart contract node est disponible sur notre GitHub publique : SmartContractNode

Configuration

L'api Kalima est fournie sous la forme d'un JAR, il en existe deux versions :

- Kalima.jar ➔ Contient les éléments nécessaires pour créer un nœud Kalima capable de se connecter sur une Blockchain Kalima, d'effectuer des transactions sur la blockchain, et de réagir de manière évènementielle à la création de nouvelles transactions (cela permet donc de créer des smarts contrats Java)
- KalimaSC.jar ➔ Inclue en plus le ContractManager qui offre la possibilité de lancer des smarts contrats Javascripts et Python

Pour utiliser l'api Kalima dans votre projet, il vous suffit donc d'inclure le jar de votre choix dans vos dépendances.

Pour cet exemple, nous prendrons donc KalimaSC.jar.

Contract Manager

L'API KalimaSC, contrairement à l'API Kalima, embarque un module ContractManager.

Ce module permet de détecter l'arrivée ou la modification de nouveaux contrats, puis les décrypte, les vérifie, et les installe. Il permet également de lancer des smart contrats selon les règles de notre choix.

Si vous avez utilisé notre lien d'inscription aux tutos (https://inscription.tuto.kalimadb.com/airdrop), vous avez reçu un mail contenant des adresses dans la blockchain tu type : /username/addr1. Pour réaliser ce tuto, il vous suffira de remplacer « username » pour matcher avec vos adresses.

Pour cet exemple, nous allons créer un smart contract node qui exécutera un smart contrat Javascript à l'arrivée de nouvelles transactions sur l'adresse /username/addr1, et un smart contrat python a l'arrivée de nouvelles transactions sur l'adresse /username/addr3. Trois paramètres seront passés à nos contrat :

- Le message reçu sous la forme d'un KMsg, qui permettra de traiter la donnée reçue dans le smart contrat
- Le clone du nœud, qui permettra par exemple de créer des transactions, ou de lire d'autres données de la blockchain
- Un logger, permettant d'ajouter des logs dans des fichiers journalisés

Implémentation

On va détailler ici deux types d'implémentations : une implémentation basique, c'est-à-dire avec le code générique fournis ainsi qu'une implémentation « from scratch » pour permettre un développement approfondi pour mieux satisfaire à vos besoins techniques.

Implémentation basique

Vous trouverez un exemple de base prêt à l'emploi, qui vous pouvez directement importer dans Eclipse par exemple :

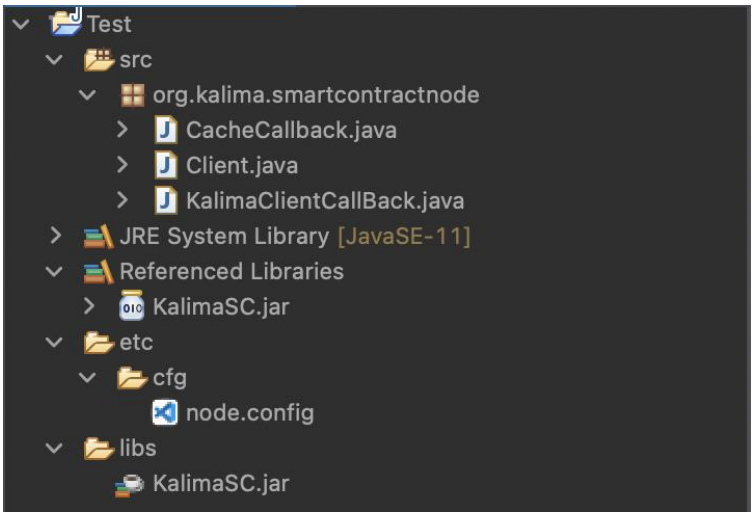
Le code comprend trois classes : une classe Client, une classe CacheCallback et une classe KalimaClientCallBack.

Lancement du nœud

Il vous faut faire une copie locale du code, puis le lancer sur votre éditeur (Attention : ce projet a été testé depuis Eclipse).

Au lancement d'Eclipse, il faut cliquer sur l'onglet « Fichier » puis « Importer » pour ouvrir votre projet local.

Il se décompose tel quel :



C'est une arborescence d'un projet Java classique, en rajoutant un fichier node.config pour le paramétrage du nœud.

Un fichier de base est fourni. Celui-ci doit être modifier en fonction de vos besoins de test avant de pouvoir lancer le code. Pour plus d'informations concernant ce fichier et comment le configurer, voir la partie « Fichier de configuration » ci-dessous.

Il y'a également une variable USERNAME dans Client.java qu'il faut modifier en fonction des adresses que vous avez reçu par mail. Vous devrez ensuite ajouter les smarts contrats nécessaires dans le répertoire git que vous avez également reçu par mail.

Une fois le nœud correctement paramétrer, on peut lancer la méthode principale avec la flèche verte « Run » en prenant soin d'insérer les arguments nécessaires :

- Le chemin du fichier de config : Si lancé tel quel, etc/cfg/node.config

On clique sur le menu déroulant à côté du bouton « Run », puis sur « Run Configurations », et enfin sur l'onglet « Arguments » pour fournir les arguments à la méthode main. On peut ensuite lancer le programme.

Fichier de configuration

Pour initialiser un nœud Kalima, il faut lui fournir quelques informations qui sont chargées depuis un fichier de configuration, exemple :

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 : Chaque nœud Kalima est composé de plusieurs « clients », et d'un « serveur ». Même si dans la majorité des cas, la partie serveur ne sera pas utilisée pour un nœud client Kalima, on doit spécifier un port pour le serveur. On peut choisir le port que l'on veut, en prenant soin de choisir un port qui n'est pas déjà utilisé sur la machine.

- FILES_PATH : Indique le dossier dans lequel seront stockés les fichiers nécessaires au fonctionnement du nœud. On y trouvera notamment les logs de l'application.

- SerialId : Pour que la connexion aboutisse, votre nœud doit être autorisé sur la blockchain. Pour initialiser la connexion, un administrateur Kalima doit vous créer une autorisation temporaire (valable 5 minutes). Cette autorisation temporaire se fait par le biais du SerialId. On peut autoriser un nœud sur une liste d'adresses, en lecture ou en écriture. Le nœud aura donc accès aux transactions de toutes les adresses sur lesquelles il est autorisé en lecture ou en écriture, en revanche il pourra créer de nouvelles transactions uniquement sur les adresses sur lesquelles il est autorisé en écriture.

- PRIVACHAIN : Nom de la privachain sur laquelle le nœud va se connecter. Pour les tutoriaux : org.kalima.tuto

Implémentation « from scratch »

On considère ici que le nœud Kalima est déjà initialisé. Sinon, il faut se référer à la documentation API_Java pour correctement paramétrer son nœud.

MemCacheCallback

On peut commencer par créer une classe qui implémente l'interface MemCacheCallback. Une instance de cette classe sera par la suite créée pour chaque adresse sur laquelle notre nœud est autorisé.

Cette classe nous permettra de réagir à l'arrivée de nouvelles transactions. C'est donc ici qui nous déciderons d'exécuter nos contrats. Pour réagir à l'arrivée de nouvelles transactions, nous implémenterons notre code dans la fonction putData qui prend 2 paramètres :

- key de type String ➔ La clé du message
- msg de type KMessage ➔ Le message reçu

La fonction runFonction de l'objet ContractManager permet d'exécuter un smart contrat, elle prend deux paramètres obligatoires :

- Le nom du contrat sous la forme d'un String.

- Le nom de la fonction que l'on veut lancer à l'intérieur du contrat. Dans notre cas, nous lancerons toujours la fonction « main ».

On peut ensuite ajouter autant de paramètres que l'on souhaite. Ces paramètres seront passés à la fonction que l'on souhaite exécuter dans le contrat. Dans notre cas, nous passerons le message reçu, le clone et un logger, comme expliqué plus haut.

Les smarts contrats peuvent être dans différents répertoires git, et les informations relatives à ces contrats publiées dans différentes adresses de la blockchain, commençant toutes par /Kalima_ Contracts (exemple : /Kalima_Contracts/Kalima). Cela permet par exemple de séparer les contrats par utilisateurs, ou selon vos propres règles. Les informations stockées dans ces adresses permettent notamment de télécharger, vérifier et décrypter les contrats.

Cela donnera donc :

```
if(kMsg.getAddress().equals(client.getContractCache())) {
	client.getClientCallBack().getContractManager().downloadContract(kMsg.getPro
ps().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);
}
```

Vous trouverez l'exemple complet de cette implémentation ici :

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

ClientCallback

Ensuite nous pouvons créer une classe qui implémente l'interface ClientCallback. Deux fonctions vont nous intéresser dans cette interface : onNewCache et onCacheSynchronized.

Au démarrage, le nœud se synchronise avec les master nodes. C'est-à-dire qu'il va recevoir les données sur les adresses auxquelles il est autorisé.

Quand une donnée arrive sur une nouvelle adresse, la fonction onNewCache est appelée. Nous pourrons donc créer nos instances de CacheCallback dans cette fonction :

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

Lorsqu'un cache est synchronisé, c'est-à-dire lorsque toutes les données d'une adresse ont été reçues, la fonction onCacheSynchronized est appelée. De plus, les informations relatives aux contrats sont stockées à l'adresse /Kalima_Contracts/… . Nous allons donc utiliser cette fonction pour initialiser notre ContractManager lorsque nous avons toutes les informations relatives aux contrats. On va également télécharger tous les contrats :

```
if(address.equals(client.getContractCache())) {
	contractManager = new ContractManager(logger, logger.getBasePath(),
new ContractCallback() {


		@Override
		public Properties getContractInfos(String key) {
			System.out.println("get key " + key);
			KMsg contractInfosMsg =
client.getClone().get(client.getContractCache(), 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());
	}
}
```

Vous trouverez l'exemple complet de cette implémentation ici :

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

Client

Enfin, nous pouvons créer notre classe principale, que nous appellerons ici « Client » et qui initialisera le nœud :

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

A ce stade, votre nœud Java est complet, cependant il fait appel à deux smarts contrats qui n'existent pas (addr1.js et addr3.py). Dans le chapitre suivant, nous détaillons comment créer un smart contrat Javascript et un smart contrat python. Pour les smarts contrats python, il sera nécessaire d'installer GraalVM et de lancer votre programme avec le SDK java inclus dans GraalVM.

Développer des contrats

Javascript

Voici un exemple de smart contrat Javascript qui crée une transaction dans /username/addr2 lorsque la température reçue est supérieure à 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
})
```

Dans les smart contrats Javascript, vous pouvez utiliser des objets Java, c'est ce que nous faisait via la première ligne, pour utiliser l'objet String de java.

Comme nous l'avons vu plus haut, notre nœud java va lancer des fonctions dans notre contrat. Il faut donc déclarer des fonctions (ici : main) et les rendre accessibles depuis le nœud java, en utilisant la syntaxe en fin de fichier : ({ id : functionName, id2 : functionName2, …)}

Vous pouvez passer les paramètres que vous souhaitez à votre fonction. Ici nous passons le message reçu pour pouvoir analyser son contenu et donc vérifier la température, ainsi que l'objet clone pour pouvoir créer ou lire des transactions par exemple, ainsi que le logger pour éventuellement créer des logs.

Python

Voici l'équivalent en 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
})()
```

On peut également utiliser des objets Java, comme on peut le voir avec les deux premières lignes.

Vous devez également déclarer vos fonctions et les exporter, mais la syntaxe est un peu différente.

Un paramètre dont nous n’avons pas besoin s’ajoute avant les paramètres que vous passez à votre fonction, vous pouvez utiliser _ pour ne pas lui donner de nom.

Déploiement

Pour déployer vos smart contrats il suffit de les push sur le répertoire git qui vous a été envoyé par mail (sous le dossier contracts). Une chaine DevOps se charge du reste. La procédure peut prendre quelques minutes.

Exécution des contrats python

Pour lancer les contrats javascript il vous suffit de lancer votre programme java normalement, pour les contrats python il faut installer graalvm (actuellement la version 22.3.1 car il y a un problème sur la dernière) : https://github.com/graalvm/graalvm-ce-builds/releases

Depuis 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
```

Exporter les variables JAVA_HOME et PATH (cf https://www.graalvm.org/latest/docs/getting-started/linux/)

Puis

```
gu install python
```

Ensuite lancer votre programme avec le SDK java inclus avec Graal, si vous avez exporter la variable JAVA_HOME :

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