Smart contract Java

Prérequis

Pour utiliser l'API Java Kalima, il est recommandé d'avoir préalablement lu la documentation API_Kalima et d'avoir installé le JDK java dans sa version 11 au minimum.

Un exemple complet de smart contract Java est disponible sur notre GitHub publique : KalimaJavaExample.

Afin de créer des smarts Contrats Java, il vous suffit juste l'API **Kalima.jar,** donc vous devriez implémenter cette API dans votre projet afin de pouvoir créer des smarts contrats Java.

1- Smarts contrats ?

Un Smart Contrat, ou contrat intelligent, est un programme informatique qui automatise l'exécution d'un ensemble d'instructions prédéfinies lorsque des conditions préalables sont réunies.

Durant son exécution, toutes les étapes de validation sont enregistrées au niveau de la blockchain. Ces enregistrements assurent la sécurisation de l'ensemble des données en empêchant leur modification ou leur suppression.

Smarts contrats Java == « Memcachecallback » que nous verrons par la suite

Les smarts contrats Java sont sous la forme d'une class java qui implémente le callback MemCacheCallback. Ce callback est implémenté pour réagir à certains évènements comme la création d'un nouveau cache ou encore l'arrivée de nouvelles transactions.

MemCacheCallback

On peut ajouter un MemCache callback par adresse pour réagir à l'arrivée de nouvelles transactions.

L'interface MemCacheCallback contient les callbacks suivants :

- getAddress : Doit retourner l'adresse correspondante
- putData : Ce callback est appelé à l'arrivé de nouvelle donnée en cache (ajout, mise à jour d'une valeur courante)
- removeData : Ce callback est appelé lorsqu'une valeur courante est supprimée

Exemple

Pour exemple, nous avons les adresses suivantes :

- /sensors : Contiendra des valeurs de capteurs, comme des températures par exemple

- /alarms/fire : Contiendra des alarmes incendies

On souhaite qu'une alarme incendie soie crée lorsqu'une température dépasse 100°C.

On souhaite également afficher dans la console l'arrivée de nouvelles températures ainsi que l'arrivée d'alarmes incendie.

Lorsque l'on relance notre node, on ne souhaite pas que les données déjà présentes soit traités à nouveau, pour éviter les doublons d'alarmes.

On commence par implémenter deux MemCacheCallbacks différents (un pour l'adresse /sensors, un pour l'adresse /alarms/fire) :

```
public class SensorsCallBack implements MemCacheCallback{

	private String address;
	private Clone clone;

	public SensorsCallBack(String address, Clone clone) {
		this.address = address;
		this.clone = clone;
	}

	@Override
	public void putData(KMessage kMessage) {
		KMsg msg = KMsg.setMessage(kMessage);
		String key = msg.getKey();
		System.out.println("new sensor value key=" + key + " body=" +
new String(msg.getBody()));
		if(key.equals("temperature")) {
			int temperature = Integer.parseInt(new
String(msg.getBody()));
			if(temperature >= 100) {
				clone.put("/alarms/fire", "temperature",
("Temperature too high: " + temperature + " °C").getBytes());
			}
		}
	}
```

```
	@Override
	public void removeData(KMessage kMessage) {

	}

	@Override
	public String getAddress() {
		return address;
	}
}
```

```
public class AlarmsCallback implements MemCacheCallback {

	private String address;

	public AlarmsCallback(String address) {
		this.address = address;
	}

	@Override
	public String getAddress() {
		return address;
	}

	@Override
	public void putData(KMessage kMessage) {
		KMsg kMsg = KMsg.setMessage(kMessage);
		System.out.println("new alarm key=" + kMsg.getKey() + " body="
+ new String(kMsg.getBody()));
	}

	@Override
	public void removeData(KMessage kMessage) {

	}
}
```

Comme on peut le voir, nos deux implémentations sont relativement simples. SensorsCallback se contente d'afficher le body des messages reçues, puis, si la clé de la donnée reçue est « temperature », il va contrôler la température, et créer une nouvelle transaction à l'adresse /alarms/fire si la température est supérieure ou égale a 100°C.

AlarmsCallback se contente d'afficher le body des données reçues.