Projet de fin de semestre
Moteur d'exécution de workflows

Kimmeng LY

Exercice 2

On ne peut pas utiliser le nom de la méthode en tant qu'identifiant car il n'est pas unique c'est-à-dire que deux méthodes d'une même classe peuvent avoir le même nom si elles ont des signatures différentes.

Exercice 3

Pour éviter les incohérences, on va exécuter les tâches par leur niveau de dépendence, c'est-à-dire que si une tâche est dépendante de deux tâches avant de pouvoir s'exécuter alors elle sera de niveau 2. Les tâches indépendante sont de niveau 0 et seront les premières à s'exécuter.
Algorithme de la méthode **execute** :

- Créer une instance de **JobValidator** avec le job renseigné lors de l'appel au constructeur de **JobExecutorSequential**, qui nous permet de valider le job et ainsi de récupérer le graphe des tâches à exécuter pour le job donné
- Créer une **Map<String, Object> results** qui nous servira à stocker les résultats de chaque tâche ainsi qu'un compteur de niveau
- Tant que le nombre de tâche traitée est inférieur au nombre de tâche dans le graphe
    - Pour chaque tâche dans le graphe des tâches à exécuter :
        - Si la tâche appartient au niveau d'exécution actuel
            - Récupérer la méthode correspondant à la tâche
            - Créer une **List<Object> params** pour stocker les valeurs des paramètres qui seront récupérés par la suite
            - Pour chaque paramètre de la méthode :
                - Si le paramètre est annoté **LinkFrom** alors :
                    - Récupérer la valeur du paramètre dans la map *results* des résultats des tâches déjà calculées dont la clé correspond à la valeur du paramètre de l'annotation **LinkFrom**
                    - Ajouter la valeur récupérée dans la liste *params*
                - Si le paramètre est annoté **Context** alors :
                    - Récupérer la valeur du paramètre dans la map *context* du job dont la clé correspond à la valeur du paramètre de l'annotation **Context**
                    - Ajouter la valeur récupérée dans la liste *params*
            - Exécuter la méthode avec la liste des paramètres récupérés
            - Ajouter le résultat de l'exécution dans la map *results*
    - Passer au niveau suivant en incrémentant le compteur de niveau
- Retourner la map *results* des résultats de calculs de toutes les tâches.

Exercice 4

Avec le multithread, on fait exécuter toutes les tâches en même temps. On assure le parallélisme des tâches indépendantes en faisant exécuter normalement les tâches non annotées **LinkFrom**. On fait *wait* une tâche seulement si celle-ci possède au moins un paramètre annoté **LinkFrom** et que la tâche dont elle dépend n'a pas encore terminée son exécution.

Exercice 5

Question 1

On définit une interface **public interface JobExecutorRemote extends Remote** qui contiendra notre méthode **Map<String, Object> execute(Job job)** qui exécute en parallèle les tâches dans le *job* passé en paramètre. L'interface doit contenir toutes les méthodes qui seront susceptibles d'être appelées à distance.

Ensuite, on définit une classe **public class JobExecutorParallelRemote extends UnicastRemoteObject implements JobExecutorRemote**. Cette classe correspond à l'objet distant. Elle doit donc implémenter l'interface définie et contenir le code nécessaire.

Pour instancier l'objet et l'enregistrer dans le registre, on va écrire dans le main de la classe **JobTrackerCentral** les opérations nécessaires pour déployer l'objet sur le serveur.
La méthode **execute()** de la classe **JobExecutorRemoteCentral** appelera cette méthode main, récupérer l'objet enregistré précédemment et appelera la méthode **execute(job)** qui renverra les résultats des calculs de chaque tâche.

Question 2

On a choisi l'API de communication **RMI** car c'est le plus simple à mettre en place pour tout ce qui est appel de methode distant.

Question 3

On doit faire l'hypothèse que les objets du contexte sont tous **Serializable**.

Question 4

On crée une **Map<String,Integer> tasksTreated** sur le serveur qui va stocker le nombre de tâche traitée et une méthode **getTasksTreated(String taskName)** qui retourne le nombre de tâche traitée du job.