

Prototypage d'un moteur d'exécution de workflow orienté données

**Travail de Bachelor**

Non Confidentiel

**Département :** TIC

**Filière :** Informatique et systèmes de communication

**Orientation :** Informatique logicielle

Peronetti Eric

23 février 2024

Travail proposé par :

Prénom Nom de la personne confiant l’étude

Nom de l’entreprise/institution

Adresse, NPA Ville

Supervisé par :

Chapuis Bertil

# Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d’études, en vue de l’obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d’auteur.

HEIG-VD

Le Chef du Département

Yverdon-les-Bains, le

## Chapitre 1 Cahier des charges

### Introduction

Ce projet de Bachelor vise à concevoir et développer un moteur d'exécution de workflow orienté données, dédié à l'optimisation des processus d'extraction, de transformation et de chargement (ETL). Ce moteur se concentrera sur l'automatisation des tâches répétitives et complexes associées à la manipulation de données.

Les principales technologies utilisées seront :

* Quarkus pour la partie backend avec GraalVM (JIT) pour faire de l’exécution de code (dans un autre langage de programmation) au runtime.
* React avec React Flow pour ce qui est du frontend et de l’affichage de nœuds de notre workflow.

### Objectifs

Les objectifs sont les suivants :

* Conception d'un modèle de données pour représenter les workflows :
  + Développer un modèle de données robuste pour représenter les workflows sous forme de graphes dirigés acycliques (DAG).
  + Modéliser les nœuds et les arcs pour une représentation précise des dépendances et des opérations au sein du workflow.
  + Implémenter une gestion efficace des entrées et sorties pour les différents nœuds du workflow.
* Développement d'un moteur d'exécution :
  + Mettre en œuvre le parallélisme pour une exécution optimisée et accélérée des workflows.
  + Incorporer un système de contexte permettant de partager des ressources au sein du workflow.
  + Établir une gestion robuste des erreurs et des exceptions pour assurer la fiabilité du moteur d'exécution.
  + Incorporer un système de mise en cache et d'évaluation paresseuse pour améliorer les performances et l'efficacité du workflow.
  + Intégrer le support pour plusieurs langages de programmation (dans un premier temps seul le Javascript sera supporté) afin d'accroître la flexibilité dans le développement des workflows.
* Création d'une interface web :
  + Concevoir une interface web intuitive pour l'édition et l'exécution des workflows.
  + Permettre aux utilisateurs de créer, de modifier et d'exécuter des workflows de manière visuelle et interactive.

### Livrables

Les livrables du travail de Bachelor sont les suivants :

* Un rapport intermédiaire
* Un rapport final
* Un résumé publiable
* Une affiche
* Un site web servant à l’édition et la visualisation des workflows
* Un backend pour traiter l’exécution des workflows

### Déroulement du travail

Voici la liste des étapes du travail :

1. Familiarisation avec les technologies utilisées (React / React flow, Quarkus et GraalVM en particulier)
2. Revue de l’existant (Github actions, AWS pipeline, …)
3. Mockup du site web / Architecture haut niveau des différentes parties (frontend et workflow engine)
4. Réalisation d’une structure de donnée convenable pour la représentation d’un workflow et de ses nœuds
5. Recherche pour le workflow engine
   1. Gestion des erreurs
   2. Système de contexte pour partager des ressources (un fichier par exemple qui pourrait être traité par plusieurs nœuds différents)
   3. Mise en cache (couplé avec l’évaluation paresseuse du workflow)
   4. Parallélisation de l’exécution des nœuds
6. Architecture du workflow engine
7. Réalisation du frontend
8. Lien entre le backend et le frontend

**Table des matières**

[Préambule I](#_Toc160803713)

[Chapitre 1 Cahier des charges II](#_Toc160803714)

[1.1 Introduction II](#_Toc160803715)

[1.2 Objectifs II](#_Toc160803716)

[1.3 Livrables II](#_Toc160803717)

[1.4 Déroulement du travail II](#_Toc160803718)

[Résumé 1](#_Toc160803719)

[Chapitre 2 Introduction 2](#_Toc160803720)

[2.1 Gestion des entrées et sortie 2](#_Toc160803721)

[2.2 Gestion des erreurs 5](#_Toc160803722)

[2.3 Gestion du cache 6](#_Toc160803723)

[2.4 Parallélisation 6](#_Toc160803724)

[2.5 Contexte 6](#_Toc160803725)

[2.6 Structure de données 6](#_Toc160803726)

[2.7 Site web 6](#_Toc160803727)

[Chapitre 3 Conclusion 7](#_Toc160803728)

[Bibliographie 8](#_Toc160803729)

[Authentification 9](#_Toc160803730)

[Annexes 10](#_Toc160803731)

# Résumé

Cliquez ou appuyez ici pour entrer du texte.

## Introduction

Le but de ce travail de Bachelor est de réaliser un moteur d’exécution pour des workflows. Les points de réflexion principaux de ce travail seront :

* **Gestion des entrées et sortie** : Comment gérer les types qui rentrent et sortent des différents nœuds ? Comment faire pour que les types soient corrects alors qu’il est possible de changer de langage ? Que faire si un nœud ne retourne rien mais que l’on veut quand même indiquer qu’il doit s’exécuter avant un autre nœud ?
* **Gestion des erreurs** : Que faire lorsqu’un nœud est marqué comme ayant une erreur ? Stopper tout le workflow ? Continuer l’exécution au maximum ?
* **Gestion du cache** : Le principe de l’évaluation paresseuse dans notre cas est de ne pas exécuter des nœuds que l’on a déjà exécuté et que l’on sait que le résultat reste inchangé.
* **Parallélisation** : On doit déterminer pour quels nœuds l’on peut effectuer leurs tâches en parallèle. On ne peut pas exécuter en même temps le nœud parent et son enfant par exemple étant donné que l’on a des dépendances sur le résultat de l’autre nœud.
* **Contexte** : Comment gérer le fait que plusieurs nœuds peuvent accéder au même fichier par exemple et, étant donné qu’il y’a du parallélisme possible. Comment faire si plusieurs nœuds veulent accéder au même fichier en même temps.
* **Structure de données**: Il est nécessaire d’avoir une structure de données adaptée pour la représentation du workflow et de ses nœuds.
* **Site web** : Quel serait la meilleure interface possible pour un utilisateur ? Comment lui montrer ce qu’il se passe actuellement dans le workflow ? Quels sont les nœuds qui posent des problèmes ?

Toutes ses points vont être traités dans ce chapitre d’introduction.

### Gestion des entrées et sortie

**Traitement des types**

Il est nécessaire de traiter le(s) type(s) de sortie / entrée de chaque nœud pour s’assurer que, lorsque l’on passe d’un nœud Javascript à un nœud Java par exemple, la cohérence de ce qui est donné en entrée soit correct. Pour ce problème il est donc nécessaire à l’utilisateur de spécifier le type que va prendre chaque entrée et la sortie d’un nœud.

Maintenant pour vérifier que les types soient corrects l’on ne peut pas juste faire une conversion vers le type que l’on veut. Si on essaie maintenant de convertir une liste d’entier vers une liste de chaîne de caractères, Java va nous laisser le faire.

|  |
| --- |
| Object intList = List.*of*(1, 2, 3); var stringList = (List<String>) intList; |

**Listing 1** – Conversion d’une liste de Integer en liste de String

Cependant si on essaie d’accéder à un élément de la liste en tant que chaîne de caractères, une erreur va être levée au runtime.

|  |
| --- |
| Object intList = List.*of*(1, 2, 3); var stringList = (List<String>) intList; var strValue = stringList.get(0); |

**Listing 2** – Accès à un élément de notre liste de Integer en tant que String

L’erreur levé est : class java.lang.Integer cannot be cast to class java.lang.String. Ceci est dû au principe de « Type erasure » de Java.

Dans la documentation de Java l’on peut voir cette explication : “Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded.” [1]

Le type de la liste à bel et bien été effacé lors de nos tests et, par ce fait, on ne peut pas utiliser cette technique pour vérifier la correspondance des types.

Une technique valide serait de vérifier récursivement les types. Par exemple, si l’on veut vérifier si le type de retour d’un nœud est bien une liste de chaîne de caractère, la première chose à faire est de vérifier si on a bien affaire à une liste. Si non, on va dire que les types ne sont pas compatibles. Si oui et que notre liste n’est pas vide, on va vérifier que le type de cet objet est bien un entier et si la liste est vide, on va juste retourner que les types sont compatibles (une liste vide d’entier est la même qu’une liste vide de chaîne de caractères).

Si on attend un objet à un quelconque endroit, on va retourner que les types sont compatibles (seulement si tout est compatible jusqu’à cet endroit-là).

|  |  |  |
| --- | --- | --- |
| Type à tester | Type attendu | Valide ? |
| List<String> | List<Integer> | Non |
| List<String> | List<Object> | Oui |
| List<String> | Object | Oui |
| List<List<String>> | List<Object> | Oui |
| Integer | Float | Non |
| Integer | Object | Oui |

**Tableau 1** – Exemple de compatibilité entre types

Voici un exemple de code qui pourrait tester la compatibilité des types (Ici on a remplacé les listes par des collections pour être le plus générique possible) :

|  |
| --- |
| enum Types {  *Collection*,  *Map*,  *Integer*,  *String*,  *Boolean*,  *Byte*,  *Short*,  *Long*,  *Float*,  *Double*,  *Character*,  *Object* }  static boolean inspect(Object obj, ArrayDeque<Types> types) {  if (types.isEmpty() || obj == null) {  return true;  }  switch (types.removeFirst()) {  case *Collection* -> {  if (obj instanceof Collection<?> list) {  if (!list.isEmpty()) {  return *inspect*(list.stream().findFirst().get(), types);  }  return true;  }  }  case *String* -> {  return obj instanceof String;  }  case *Character* -> {  return obj instanceof Character;  }  case *Boolean* -> {  return obj instanceof Boolean;  }  case *Integer* -> {  return obj instanceof Integer;  }  ...  case *Object* -> {  return true;  }  }  return false; } |

**Listing 3** – Exemple de fonction pour le test de compatibilité

On peut donc tester ce code avec :

|  |
| --- |
| var result = *inspect*(  List.*of*(List.*of*(1, 2, 3)),   new ArrayDeque<>(List.*of*(Types.*Collection*, Types.*Collection*, Types.*Integer*)) ); |

**Listing 4** – Exemple d’utilisation de la fonction pour le test de comptabilité

Ici la variable *result* vaudra *true*. On attend une collection de collection d’entier et on teste avec *List<List<Integer>>* qui est valide pour ce type.

**Traitement des nœuds qui ne retournent pas de valeur**

Imaginons que l’on ait les nœuds avec les fonctions suivantes : accès à un fichier, calcul de la longueur d’un fichier, ajout à la fin d’un fichier et écriture dans les logs.

Une image contenant texte, écriture manuscrite, Police, ligne

Description générée automatiquement

**Figure 1** - Exemple de problème avec des nœuds qui ne retournent pas de valeur

En vert les entrées et en rouge les sorties

Ici, le problème est que l’on ne sait pas à quel moment le nœud *Append* va s’exécuter (avant ou après le nœud *Get length*). Et actuellement il n’y a pas de moyen de dire que *Get length* doit s’exécuter après *Append*.

Une possibilité serait de dire que tous les nœuds qui n’ont pas de type de retour auront quand même la possibilité d’avoir une sortie (avec un type *Flow* par exemple) et rajouter une entrée à tous les nœuds pour cette sortie particulière.

Une image contenant texte, écriture manuscrite, ligne, Police

Description générée automatiquement

**Figure 2** – Correction du problème des nœuds qui retournent aucune valeur

En vert les entrées, en rouge les sorties et en jaune les entrées pour indiquer l’ordre de déroulement

Maintenant, *Get length* ne sera exécuté que quand *Append* sera terminé. Même si le nœud *Append* change de type de sortie et maintenant décide de retourner le nombre de charactères qui viennent d’être ajoutés au fichier, cette technique fonctionne toujours car l’entrée que l’on a rajoutée ne dépend pas du type de valeur mais est là seulement pour gérer l’ordre de déroulement du workflow.

Une image contenant texte, écriture manuscrite, ligne, Police

Description générée automatiquement

**Figure 3** – Exemple avec des nœuds qui retournent aucune/une valeur

En vert les entrées, en rouge les sorties et en jaune les entrées pour indiquer l’ordre de déroulement

**Dans cet exemple, on va tout d’abord avoir le nœud *File* qui va s’exécuter, puis le nœud *Append* puis, en parallèle les nœuds *Get length* et *Log* (celui juste après *Append*) et, quand les deux nœuds précédents ont terminé, le tout dernier *Log*.**

**Avec cette manière de faire, il n’y aura plus de problèmes pour indiquer comment doit s’exécuter le workflow.**

### Gestion des erreurs

Pour la gestion d’erreur il y’a plusieurs manières de procéder lorsqu’une erreur survient :

* Arrêter l’exécution de tout le workflow instantanément.
* Laisser continuer l’exécution du workflow au maximum (c’est-à-dire jusqu’à ce que le nœud qui est en erreur soit nécessaire). Peut être intéressant lorsque l’on met en [cache](#_Gestion_du_cache) les sorties des nœuds.

Les deux solutions ont leurs avantages. Si l’on arrête l’exécution du workflow directement, l’utilisateur pourra directement modifier les nœuds et, contrairement à la deuxième solution, ne perdra donc pas de temps à attendre devant son ordinateur en attendant que le workflow se termine alors qu’il sait déjà que l’entièreté du workflow ne sera pas exécutée. La deuxième solution peut être utile si l’utilisateur laisse tourner le workflow pendant 10 minutes et reviens pour constater qu’il y’a une erreur. Étant donné que la sortie des nœuds est [mise en cache](#_Gestion_du_cache), il est donc pratique que l’exécution continue au maximum.

Dans la figure qui suit l’on peut voir un exemple d’exécution du workflow au maximum. L’erreur est survenue au nœud 4 mais ici on peut voir que les nœuds 5 et 7 ont quand même été exécutés.

Une image contenant diagramme, Post-it, ligne

Description générée automatiquement

**Figure 4** – Exemple d’un workflow lors duquel l’on continue l’exécution au maximum

En vert les nœuds qui sont exécutés et en rouge les nœuds qui ont une/des erreur(s)

La solution retenue est un mélange des deux solutions. Lorsqu’une erreur survient dans un nœud, l’exécution continue au maximum. Le changement est que, dès que les nœuds sont en erreur / terminés, il est possible de modifier le code et les types d’entrée / sortie de ces nœuds. Il sera aussi possible de changer les liens entre les nœuds mais, les liens qui touchent à des nœuds encore en cours / attente d’exécution ne seront pas éditable.

Une image contenant diagramme, ligne, Post-it, Police

Description générée automatiquement

**Figure 5** – Exemple d’un workflow en cours d’exécution

En bleu les nœuds qui sont en cours d’exécution, en vert les nœuds qui sont exécutés et en rouge les nœuds qui ont une/des erreur(s). Les flèches / bordures en rouge démontrent l’impossibilité de modification de l’élément

### Gestion du cache

**Quand mettre les sorties en cache**

Dans cette partie, on veut faire en sorte que, lorsqu’on exécute notre workflow, la sortie des nœuds soit sauvegardée pour être réutilisée lorsque l’on va exécuter le workflow à nouveau.

Un premier problème avec ce cache, est que ceci ne peut marcher seulement si les nœuds sont dits « purs ». Le principe est le même que pour une fonction pure [2]. Si on a un nœud qui n’est pas considéré pur (écriture dans un fichier par exemple), alors la sortie ne pourra pas être stockée dans le cache pour ce nœud.

Il y’a plusieurs cas à considérer :

* Si le nœud écrit dans un fichier (utilise un objet fichier en mode écriture), le considérer comme impur
* Si le nœud n’est pas marqué comme pur, le considérer comme impur (l’utilisateur pourra marquer un nœud comme pur)
* Si le nœud lit dans un fichier et que son contenu a changé, le considérer comme « dirty »
* Si le code du nœud à été modifié, le considérer comme « dirty »

Un nœud marqué « dirty » est un nœud pur qu’il va falloir exécuter à nouveau et stocker le résultat dans le cache. Il est possible que la prochaine fois le nœud ne soit plus marqué comme « dirty » et donc que l’on puisse utiliser la valeur dans le cache. Alors que les nœuds impurs ne stockent jamais leurs sorties dans le cache.

Une image contenant texte, Police, écriture manuscrite, ligne

Description générée automatiquement

**Figure 6** – Exemple d’un fichier qui a changé de contenu

En haut à droite des nœuds on a : \* (« dirty »), P (pur), I (impur). Si le caractère est en rouge cela veut dire que le choix a été forcé par le système et en noir c’est un choix de l’utilisateur.

Dans cet exemple, le fichier a changé de contenu (on part du principe qu’ici, le nom du fichier n’a pas changé). On peut voir que le hash du contenu du fichier stocké en cache n’est plus le même. Notre nœud *File* est donc marqué comme « dirty » (il n’est ni pur ni impur). On peut voir ici que *Get length* est un nœud pur parce que si le fichier n’a pas changé (c’est-à-dire que le hash du fichier est aussi resté le même), la sortie sera la même. Le nœud *Log* est impur parce que l’on veut écrire dans les logs à chaque exécution.

Une image contenant texte, écriture manuscrite, Police, ligne

Description générée automatiquement

**Figure 6** – Exemple d’ajout dans un fichier

En haut à droite des nœuds on a : \* (« dirty »), P (pur), I (impur). Si le caractère est en rouge cela veut dire que le choix a été forcé par le système et en noir c’est un choix de l’utilisateur.

Ici, le nœud *File (Write)*[[1]](#footnote-1)sera aussi considéré comme pur parce qu’il est possible que le nœud soit utilisé par un *Get length* et que son contenu n’ait pas changé entre deux exécutions. Il sera dont traité de la même manière que *File* comme démontré précédemment. Le nœud *Append* est lui considéré comme impur parce qu’il prend en tant qu’entrée un *File (Write)*. Si l’entrée de *Append* avait été de type File (et que l’on relie quand même un *File (Write)*)alors à ce moment-là le nœud n’aurait pas forcément été impur (choix de l’utilisateur).

En résumé :

* Seul un nœud **pur** et **non marqué comme « dirty »** peut aller récupérer sa sortie dans le cache
* Les nœuds **purs** stockent leurs sorties dans le cache

**Que mettre en cache et comment**

Un autre souci que pose la mise en cache est la taille des éléments. Si on travaille maintenant avec des fichiers qui font plusieurs Go, comment faire si, dans le premier nœud où l’on peut écrire du code, on retourne une liste de chaîne de caractère qui contient toutes les lignes du fichier texte. Comment faire pour stocker dans le cache cette sortie ?

Une solution serait de stocker la sortie de chaque nœud dans un fichier (en JSON par exemple). Il n’y aurait donc pas de risque d’avoir trop de mémoire utilisée. Le chargement du cache sera effectivement plus lent mais il n’y aura pas de risque d’utiliser toute la mémoire disponible parce que l’on stockait dans un objet toutes les sorties des nœuds.

### Parallélisation

### Contexte

### Structure de données

### Site web

.

## Conclusion

Cliquez ou appuyez ici pour entrer du texte.

# Bibliographie

|  |  |
| --- | --- |
| [1] | Oracle, «Type Erasure,» [En ligne]. Available: https://docs.oracle.com/javase/tutorial/java/generics/erasure.html. [Accès le 8 Mars 2024]. |

# Authentification

Le soussigné, , atteste par la présente avoir réalisé seul ce travail et n’avoir utilisé aucune autre source que celles expressément mentionnées.

Yverdon-les-Bains, le

# Annexes

Excalidraw (pour les dessins des différents graphes): <https://excalidraw.com/>

**Glossaire**

**JIT**: Just in time

**ETL**: Extract, transform and load

**Index**

**Table des figures**

[**Figure 1** - Exemple de problème avec des nœuds qui ne retournent pas de valeur 4](#_Toc160803363)

[**Figure 2** – Correction du problème des nœuds qui retournent aucune valeur 5](#_Toc160803364)

[**Figure 3** – Exemple avec des nœuds qui retournent aucune/une valeur 5](#_Toc160803365)

[**Figure 4** – Exemple d’un workflow lors duquel l’on continue l’exécution au maximum 6](#_Toc160803366)

[**Figure 5** – Exemple d’un workflow en cours d’exécution 6](#_Toc160803367)

**Liste des tableaux**

[**Tableau 1** – Exemple de compatibilité entre types 3](#_Toc160800397)

**Liste des codes sources**

[**Listing 1** – Conversion d’une liste de Integer en liste de String 2](#_Toc160800420)

[**Listing 2** – Accès à un élément de notre liste de Integer en tant que String 3](#_Toc160800421)

[**Listing 3** – Exemple de fonction pour le test de compatibilité 4](#_Toc160800422)

[**Listing 4** – Exemple d’utilisation de la fonction pour le test de comptabilité 4](#_Toc160800423)

1. *File (Write)* est un type spécifique permettant l’écriture en plus de la lecture que *File* permet. Ce type spécifique marque directement les nœuds qui l’utilisent en tant que nœud impur. [↑](#footnote-ref-1)