Architecture logicielle
4AL2 – CC2

DOCUMENTATION FONCTIONNELLE ET TECHNIQUE

GUILLAUME TOUCHET

Table des matières

I. Les points d'entrées du programme

1. La console

La console permet d'effectuer des commandes ou des requêtes sur les données stockées dans les fichiers Json du programme à l'aide de commandes suivies de paramètres.

La première instruction entrée devra être un des mots clés de l'interpréteur, pour afficher la liste des mots clés disponibles, taper 'help'.

Si le mot clé entré ne correspond à aucune commande disponible, l'utilisateur verra un message d'erreur apparaître dans la console.

Le nombre de paramètres à renseigner à la suite de la commande dépendra de la commande entrée, certaines commandes acceptent un nombre variable de paramètres, par exemple les requêtes de données acceptent un ID en paramètre pour rechercher une entité précise, plutôt que de recevoir la liste complète des entités enregistrées.

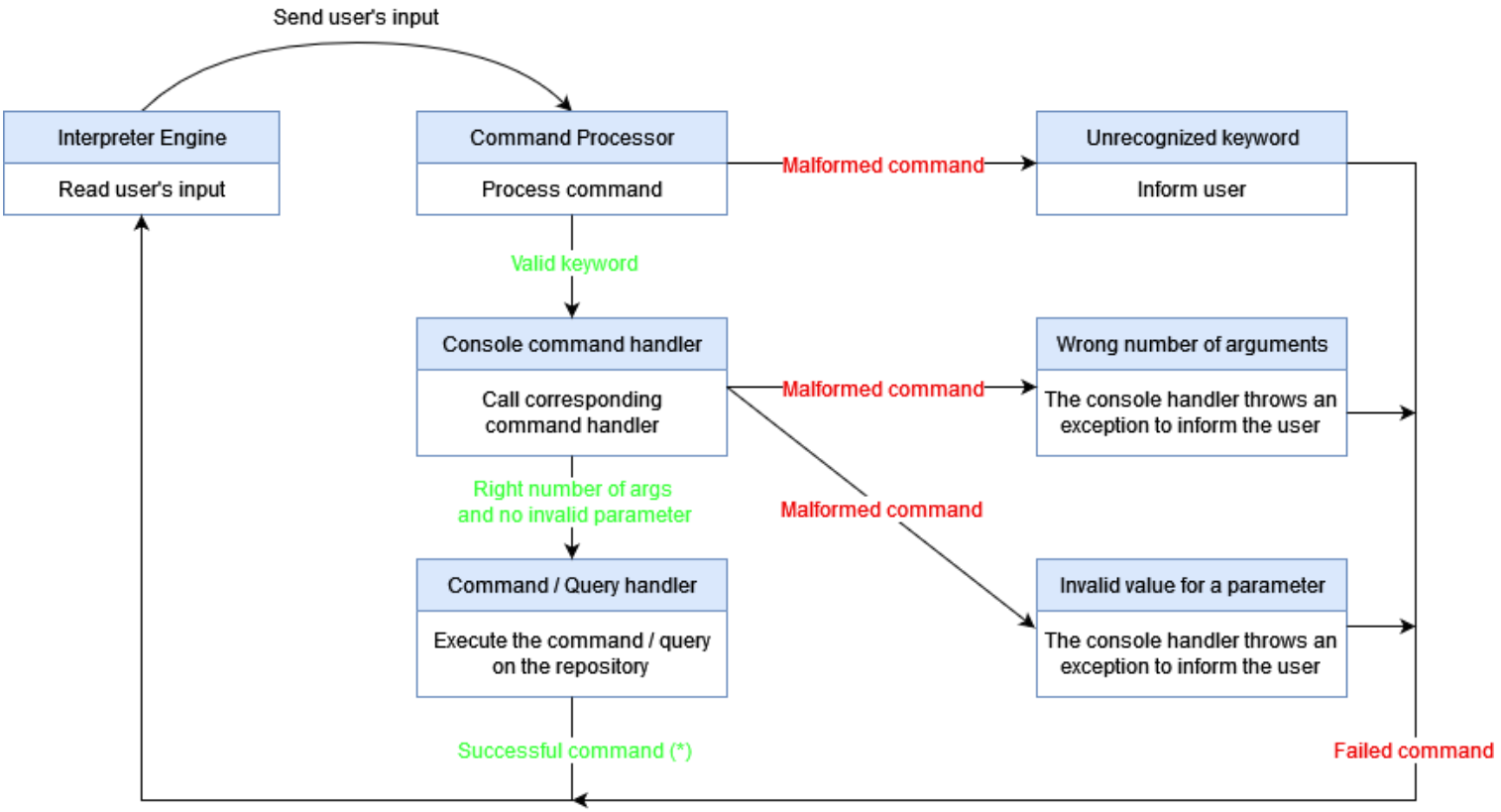
Si le nombre de paramètre ne correspond pas à celui attendu par la commande, l'utilisateur verra un message d'erreur apparaître, avec une description du fonctionnement de la commande entrée.

Si un des paramètres n'est pas valide, par exemple un texte au lieu d'un nombre renseigné pour le département d'un Worker, l'utilisateur verra le même message d'erreur que précédemment.

Pour afficher cette description sans avoir besoin de provoquer une erreur, l'utilisateur peut utiliser la commande 'help' qui accepte en paramètre supplémentaire un mot clé, par exemple 'help createworker'.

Pour quitter le programme, l'utilisateur peut entrer la commande 'quit'.

Schéma de fonctionnement de l'interpréteur de commande :



(*) There still could be repositories exceptions after this point that could make the request fail, but this schematic's point is to explain the workflow of the program's CLI, so we will ignore them for the moment

L'Interpreter enregistre la commande et les paramètres entrés par l'utilisateur, il appelle ensuite son Command Processor via la méthode 'process' en lui envoyant les données sous forme de String.

Le Processor découpe le String reçu en paramètre afin de récupérer le mot clé de la commande et les paramètres, il vérifie ensuite que le mot clé correspond bien à une commande disponible.

Si le mot clé existe, il récupère le Console Handler lié à la commande correspondante et l'instancie en lui injectant les Command ou Query Handlers instanciés nécessaires à son fonctionnement, puis appelle sa méthode Handle en lui passant les paramètres de la commande.

Le Console Handler effectue les vérifications de conformité des paramètres, puis appelle son Command ou Query Handler injecté en lui passant un Data Object contenant les données récupérées des paramètres de la commande dont il a besoin.

Toutes les informations de chaque commande (mot clé, nombre de paramètres attendus, exemple d'utilisation, Console Handler lié et Command ou Query Handlers nécessaires au fonctionnement du Console Handler) sont renseignées dans un énumérateur possédant plusieurs méthodes d'accès aux commandes enregistrées permettant de récupérer la liste complète des commandes ou une commande spécifique.

2. L’API Quarkus

Work in progress ☹

II. Les données persistantes

1. Les fichiers Json

Chaque modification apportée par l'utilisateur via l'interpréteur modifiera les données enregistrées dans les fichiers Json situés sous '/res/'.

Un fichier de backup sera aussi créé dans le dossier '/res/backups/' avant toute modification de données, afin de permettre d'effectuer un rollback en remplaçant manuellement le fichier de stockage par le dernier backup (le nom du fichier de backup contient la date et l'heure de sa création, ce qui permet de le retrouver facilement).

2. Les Data Accessors

Les Data Accessors contiennent les méthodes de lecture et d'écriture des données persistantes de l'application.

Ils sont exposés sous la forme d'une interface, afin de pouvoir être injectés dans les Repositories peu importe le type de support d'écriture et lecture de données, sans avoir besoin de modifier le fonctionnement des Repositories.

Le Json Data Accessor lit et écrit les fichiers Json de l'application, si le fichier est introuvable, il le créé. Il créé aussi automatiquement un backup du fichier avant toute modification afin de pouvoir facilement retrouver des données suite à une erreur de modification.

III. Infrastructure

1. Les Repositories

Le Data Repository permet d'accéder aux données persistantes via un CRUD, il a besoin d'un Data Accessor pour écrire des données sur le support fournit, ainsi que d'initialiser sa mémoire au lancement du programme.

Toute commande effectuée sur les données persistantes sont aussi effectuées dans la mémoire du Data Repository, ce qui permet de respecter l'isométrie entre les données persistantes et la mémoire du Repository.

Le type de données persistantes dépend du type de Data Accessor fournit (uniquement Json dans le cadre de ce projet, mais le Data Accessor pourrait permettre d'écrire dans une base de données, sans changer le fonctionnement du Repository).

Le Memory Repository stocke les données dans la mémoire du programme, le cycle de vie des données est donc limité à celui du programme, il est utilisé pour les tests afin de ne pas polluer les données réelles des fichiers Json.

A la différence du Data Repository, il n'a pas besoin de Data Accessor car il ne modifie pas de données persistantes.

2. Les Repositories Factory et Retainer

TODO

IV. Application

1. Les Services Handlers

Les Services Handlers permettent l'accès ou la modification des données des Repositories grâce à des Commands ou des Queries, via leur méthode 'handle' qui attend en paramètre un Data Object.

Le type de retour de la méthode dépend du type d'action gérée par le Handler, par exemple un Delete renverra un Boolean, un Create ou Update renverront l'entité créée ou modifiée.

Ils effectuent aussi les validations de modifications des données telles que la vérification de login utilisateur unique, de mot de passe valide, d'existence de l'entité avant d'appeler le Repository (dans le cas d'un Delete ou Read par ID par exemple), l'appel à une API externe, etc...

Ils utilisent les Builders pour effectuer des création ou modification des entités avant d'appeler un Repository (voir la partie sur les Builders).

2. Le Handlers Container

Le Handlers Container permet d'initialiser tous les Command et Query Handlers du programme grâce à son constructeur nommé 'initialize'.

La méthode static 'initialize' attend en paramètre une implémentation de Repository Factory (In Memory ou Data), ainsi qu'une instance de Password Validator et de l'API de validation de moyen de paiement (Stub).

Elle instancie un Repository pour chaque entité puis instancie chaque Command ou Query Handler en lui injectant les dépendances nécessaires à leur fonctionnement grâce à sa méthode 'register', les Handlers sont donc enregistrés dans la mémoire du Container pour une future utilisation.

Pour récupérer un Handler enregistré dans le Container, il faut appeler la méthode 'getCommandHandler' ou 'getQueryHandler', en lui passant en paramètre la classe du Handler désiré, par exemple : 'getCommandHandler(CreateProjectCommandHandler.class);'

Le container permet de n'instancier qu'une seule fois les Handlers en leur injectant leurs dépendances, et de les stocker pour un accès plus rapide aux données.

V. Domain

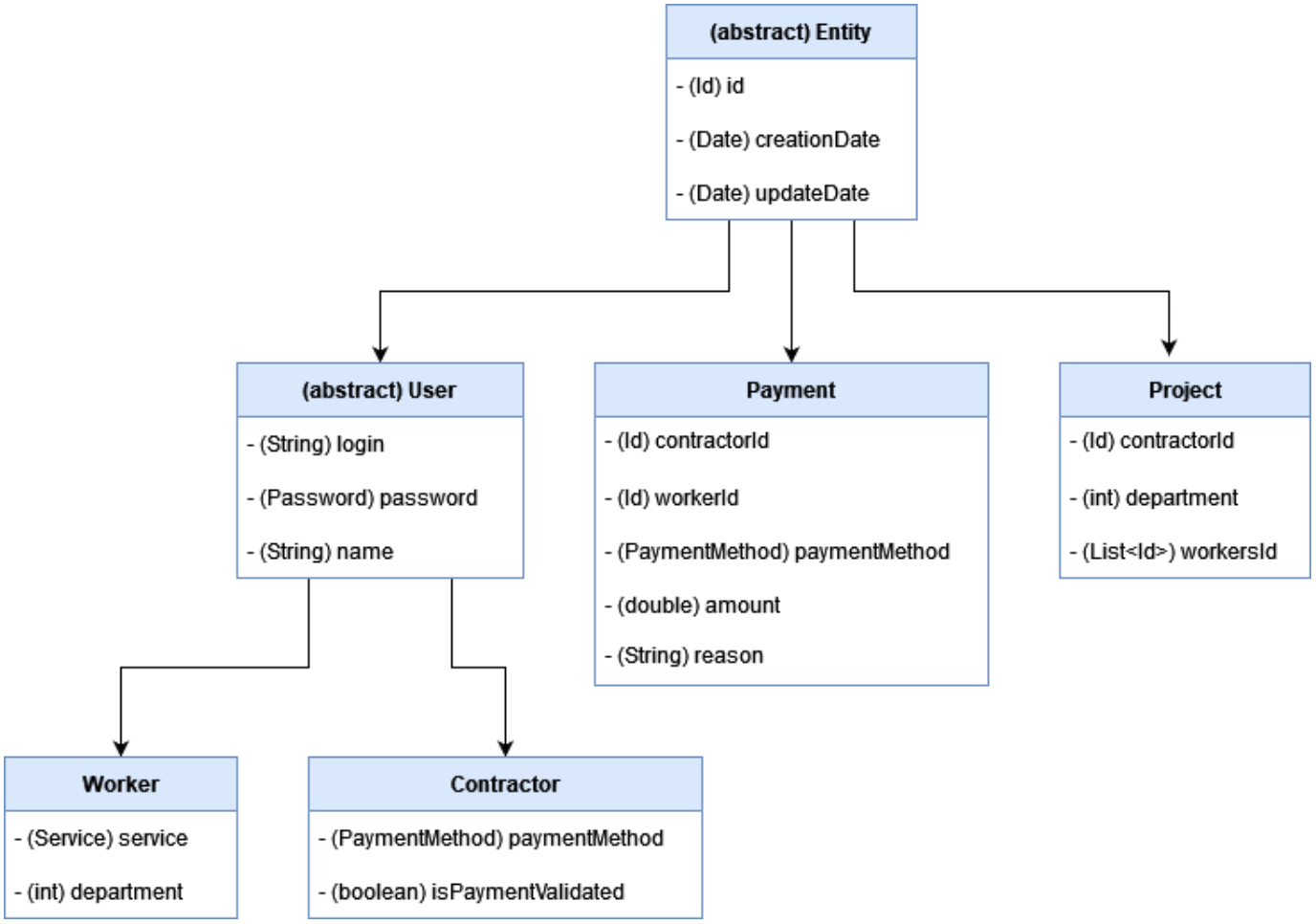
1. Les Entities

Entity :

Les Entities contiennent les données métier du programme, elles descendent toutes de la classe abstraite Entity afin de pouvoir utiliser un seul unique Repository de type générique héritant de la classe Entity. Elle contient une propriété ‘id’ utilisé pour les recherches d’entités, ainsi que deux propriétés de type Date pour enregistrer leurs dates de création et de dernière modification.

User :

La classe abstraite User hérite de la classe Entity et est héritée par toutes les classes utilisateurs (Contractor et Worker), elle contient toutes les propriétés communes des classes qui en hérite : un login sous forme de String, un mot de passe sous forme d’un Value Object ‘Password’ et un String contenant le nom de l’utilisateur.



2. Value Object Pattern

Le programme contient des Value Objects permettant de créer des instances de Wrappers :

- Id est un Wrapper de UUID, qui permet grâce à ses constructeur nommés de générer un nouvel ID ou d'en créer un depuis un String au bon format.
- Date est un Wrapper de SimpleDateFormat permettant de générer une date au format jour-mois-année-heure-minute-seconde.
- Password permet de stocker un mot de passe sous la forme d'un String.

Les Value Objects contiennent tous les méthodes de comparaison et d'accès 'equals' et 'toString'.

3. Builder Pattern

Les Builders permettent de créer une Entity ou de la modifier propriété par propriété, ils possèdent deux constructeurs nommés 'init', l'un permet de créer une nouvelle Entity en lui attribuant un nouvel Id et une date de création, laissant ses autres propriétés nulles, l'autre attend une Entity complête (Id et date de création déjà renseignés) et enregistre ses propriétés pour une future modification.

Les Setters permettent de donner une valeur à une propriété ou de la modifier avant de récupérer l'Entity.

Ils possèdent une méthode 'build' permettant de récupérer une instance de l'Entity, elle lève une exception si une ou plusieurs propriété est nulle.

VI. Exemple de modification technique

Dans cet exemple, nous allons ajouter un service permettant de récupérer une liste de Contractors en fonction de leurs moyens de paiements.

1. Création du service

Avant de créer le service, nous allons créer le Data Object contenant les données dont aura besoin notre service, ajoutons le dans le package 'application.services.dtos.contractor' et appelons le ReadByPaymentContractorQuery, il doit implémenter l'Interface Query afin d'être accepté comme type d'entrée de notre Handler. Il contiendra la méthode de paiement recherchée.

```
public final class ReadByPaymentContractorQuery implements Query
{
    public final PaymentMethod paymentMethod;

    public ReadByPaymentContractorQuery(PaymentMethod paymentMethod)
    {
        this.paymentMethod = paymentMethod;
    }
}
```

Il faut ajouter un Handler dans le package 'application.services.handlers.contractor', appelons le ReadByPaymentContractorQueryHandler.

La Classe Implémentera l'Interface QueryHandler, dont le type de retour sera une liste de Contractor, et le type de Query sera le Data Object ReadByPaymentContractorQuery.

Ce Handler aura besoin d'une dépendance : un Contractor Repository, afin de pouvoir récupérer les données.

```
public class ReadByPaymentContractorQueryHandler implements QueryHandler<List<Contractor>, ReadByPaymentContractorQuery>
{
    public final Repository<Contractor> contractorRepository;

    public ReadByPaymentContractorQueryHandler(Repository contractorRepository)
    {
        this.contractorRepository = contractorRepository;
    }

    @Override
    public List<Contractor> handle(ReadByPaymentContractorQuery query)
    {
        return null;
    }
}
```

L'implémentation de la méthode 'handle' dépend du besoin, dans notre cas, un simple filtre sur le Stream reçu du Repository.

```
@Override
public List<Contractor> handle(ReadByPaymentContractorQuery query)
{
    return this.contractorRepository.read()
            .filter(contractor -> contractor.getPaymentMethod().equals(query.paymentMethod))
            .collect(Collectors.toList());
}
```

La classe est prête, il faut l'ajouter dans le Handlers Container.

```
// Register Contractor services
handlersContainer.register(new CreateContractorCommandHandler(contractorRepository, workerRepository, passwordValidator));
handlersContainer.register(new DeleteContractorCommandHandler(contractorRepository));
handlersContainer.register(new ReadAllContractorQueryHandler(contractorRepository));
handlersContainer.register(new ReadContractorQueryHandler(contractorRepository));
handlersContainer.register(new UpdateContractorCommandHandler(contractorRepository, passwordValidator));
handlersContainer.register(new ValidatePaymentCommandHandler(contractorRepository, paymentMethodValidatorApi));
+ handlersContainer.register(new ReadByPaymentContractorQueryHandler(contractorRepository));
```

Le service est prêt, il nous reste à modifier le Command Processor le consommer.

2. Ajout à la liste de commandes

Il faut tout d'abord ajouter la commande à l'énumérateur ConsoleCommand qui stocke la liste de commandes, dans le package 'console.engine'.

Chaque commande a besoin de plusieurs informations :

- Le mot clé (String)
- Le nombre de paramètres attendu
- L'exemple d'utilisation
- Le Console Handler lié à la commande
- Le ou les types de paramètres attendus par le Console Handler (utilisés pour la Reflection)

Dans notre cas, le mot clé sera 'SELECTCONTRACTORBYPAYMENT', la commande attendra deux paramètres (le mot clé + la méthode de paiement recherchée), le Console Handler sera celui que nous allons créer dans la prochaine étape, et son constructeur attendra le Handler que nous avons créé dans la partie précédente.

```
READ_CONTRACTOR_BY_PAYMENT("SELECTCONTRACTORBYPAYMENT", 2,
        "SELECTCONTRACTORBYPAYMENT paymentMethod -> retrieve contractors by their payment methods\n",
        ReadByPaymentContractorConsoleHandler.class,
        new Class[] { ReadByPaymentContractorQueryHandler.class }),
```

La commande est prête à être appelée par l'utilisateur, mais il nous reste à implémenter son Console Handler.

3. Création du Console Handler

Pour compléter l'ajout de fonctionnalité dans le programme, nous allons créer le Console Handler, il sera dépendant du Query Handler créé précédemment qui lui sera injecté lors de son instanciation.

```
public class ReadByPaymentContractorConsoleHandler implements ConsoleHandler
{
    private final QueryHandler<List<Contractor>, ReadByPaymentContractorQuery> queryHandler;

    public ReadByPaymentContractorConsoleHandler(QueryHandler queryHandler) throws NullPointerException
    {
        this.queryHandler = Objects.requireNonNull(queryHandler);
    }

    @Override
    public void handle(String[] params) throws WrongNumberOfArgumentException
    {

    }
}
```

La méthode 'handle' va vérifier que le nombre de paramètres est bien celui attendu par la commande, puis appelle le Handler injecté pour récupérer les données et les afficher.

```
@Override
public void handle(String[] params) throws WrongNumberOfArgumentException
{
    if (params.length == ConsoleCommand.READ_BY_PAYMENT_CONTRACTOR.parameters)
    {
        try {
            List<Contractor> contractors = this.queryHandler.handle(new ReadByPaymentContractorQuery(
                    PaymentMethod.valueOf(params[1].toLowerCase())
            ));
            if (contractors.size() > 0)
            {
                contractors.forEach(System.out::println);
            }
            else
            {
                System.out.println("No Contractor registered with payment method [" + params[1].toLowerCase() + "]");
            }
        } catch (IllegalArgumentException e) {
            System.out.println("Error: unknown payment method [" + params[4] + "]");
        }
    }
    else
    {
        throw new WrongNumberOfArgumentException(ConsoleCommand.READ_BY_PAYMENT_CONTRACTOR);
    }
}
```