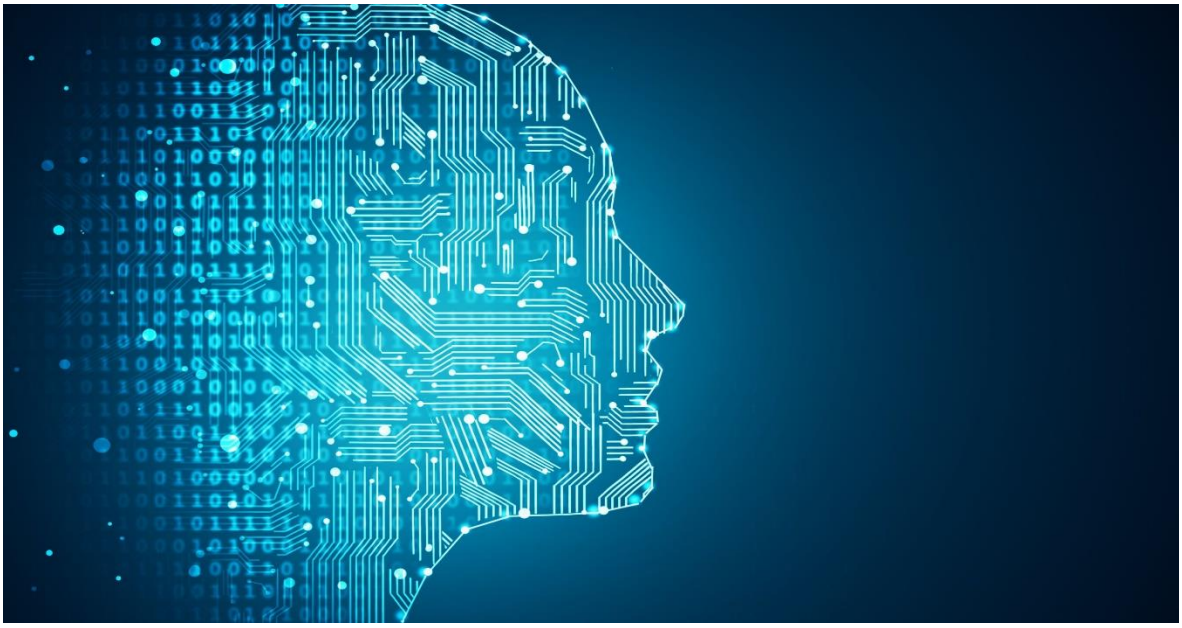
Agents Intelligents - M2 SID

Agent Internaute



Membres du groupe

Anne-Laure CHARLES, Louis MASSICARD, Claire ROMANEL, Adel YACIA

Master Miage 2 SID

IDMC Nancy

Université de Lorraine

Année 2020/2021

Rappels du sujet 1
Analyse, conception et développement 2
Architecture 2
Généralités 2
Modélisation de l'environnement 4
Base de données 5
Caractérisation des situations d’interaction 6
Définition des comportements 6
Communication entre l’interface graphique et l’agent internaute 8
Interactions avec les autres agents 10
Format d’échange des messages 10
Stratégies 11
Stratégies définies pour notre agent 11
Evaluation de la pertinence des stratégies 12
Gestion de projet 12
Outils et technologies utilisés 13
Choix effectués 13
Difficultés rencontrées 14

Rappels du sujet

L'objectif de ce projet était d'utiliser la plateforme Jade pour créer un réseau de 4 types d'agents intelligents communiquant entre eux : des agents internautes, des agents distributeurs, des agents producteurs et enfin un agent e-réputation.

Notre groupe était chargé de concevoir l'agent internaute. Nous devions mettre en place une interface graphique afin de pouvoir effectuer des actions vis-à-vis des agents distributeurs et e-réputation, les agents producteurs n'ayant par contre pas de rapport direct avec nous. L'interface graphique devait également permettre de spécifier un profil et des préférences pour chaque internaute, afin d'aider les distributeurs à nous envoyer des recommandations personnalisées.

Vis à vis de l'agent e-réputation, nous devions pouvoir envoyer des notes pour les œuvres consommées, mais également pour les distributeurs, les producteurs et les artistes (acteurs, réalisateurs, ...).

Vis à vis des agents distributeurs, nous devions pouvoir effectuer deux types de recherches :

- la recherche d'une oeuvre spécifique par nom de l'oeuvre, dans ce cas les agents distributeurs devaient nous renvoyer soit une correspondance trouvée dans leur catalogue, soit des recommandations d'oeuvres similaires s'ils ne trouvaient pas de correspondance,
- une recherche d'œuvre non spécifique avec des critères de recherche, comme par exemple un ou plusieurs genres musicaux, ou encore un film avec un acteur en particulier.

Enfin, nous devions mettre en place des stratégies pour recommander (ou non) une ou plusieurs œuvres dans les résultats de recherche envoyés par les distributeurs, ainsi qu'un moyen d'évaluer notre satisfaction.

Analyse, conception et développement

Architecture

Généralités

Nous avons deux parties à gérer. D'une part, un ou plusieurs agent(s) internaute chargé de communiquer avec les autres agents, en particulier les agents distributeurs et e-réputation. D'autre part, une partie graphique permettant à un internaute de communiquer avec l'agent internaute pour transmettre des requêtes aux autres agents de la plateforme Jade et afficher leurs réponses.

Notre projet comporte deux applications :

- une application web fournissant une interface graphique pour l'internaute (framework javascript VueJS),
- une application Java (Spring Boot et Jade) découpée en plusieurs packages:
    - un package contenant le code nécessaire pour créer un conteneur jade, des agents internautes et leurs behaviours, ainsi que des agents "mocks" nous permettant de simuler la présence des autres agents avec leurs behaviours associés et ainsi tester notre application,
    - un package contenant les stratégies,
    - des packages nécessaires à la persistance de certaines données dans une base locale et à la mise en place d'une API permettant de communiquer avec l'application web via des requêtes HTTP.

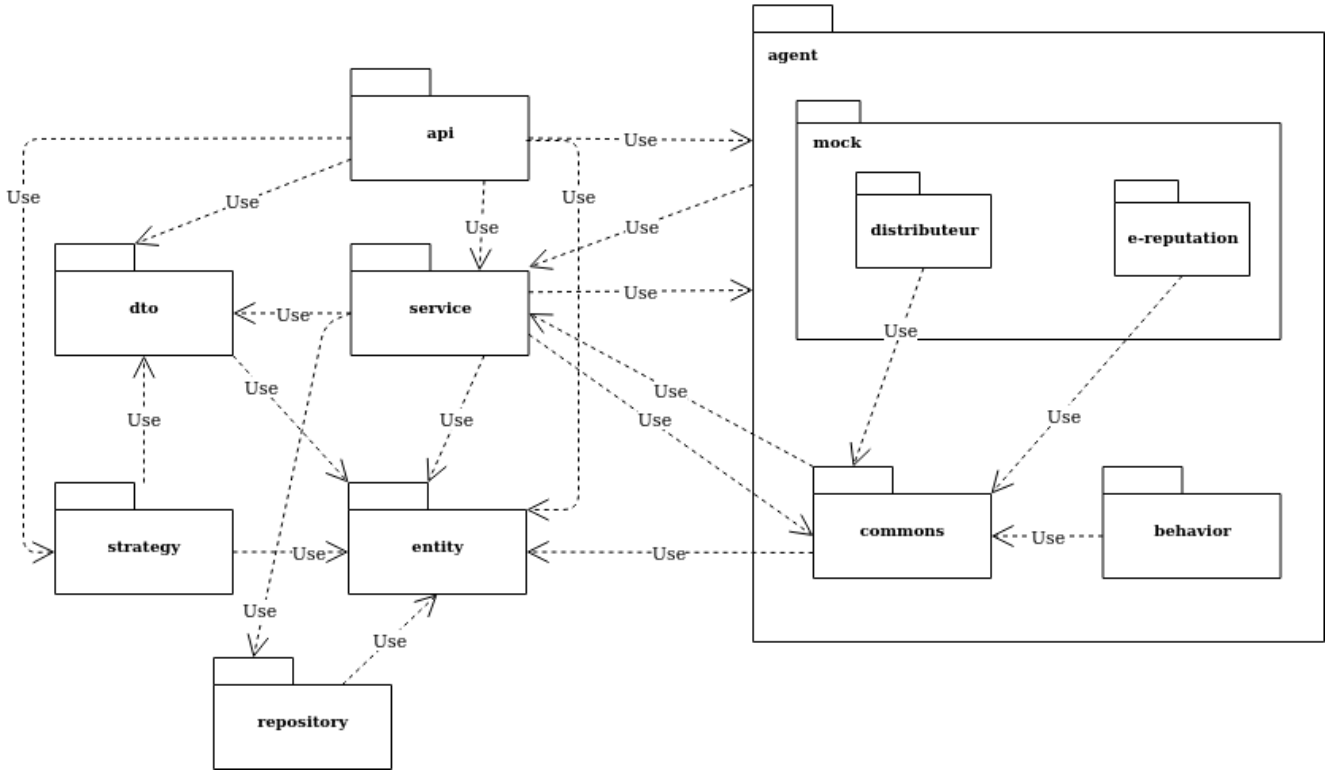


Figure 1 : diagramme de packages

Modélisation de l'environnement

Sur le schéma ci-dessus, nous pouvons voir, les différents échanges qui sont réalisés entre les agents :

- l’agent e-réputation qui communique avec l’ensemble des agents,
- l’agent distributeur qui communique avec l’ensemble des agents,
- l’agent producteur qui communique seulement avec les agents distributeurs et e-reputation,
- enfin, l’agent internaute avec les agents distributeurs et e-reputation.

Base de données

Notre application avait besoin de persister certaines données afin de pouvoir fonctionner. En particulier, nous devions conserver les informations des préférences des internautes, ainsi que les œuvres déjà achetées/consommées. Nous avons donc créé des classes “Entity” dans le package du même nom.

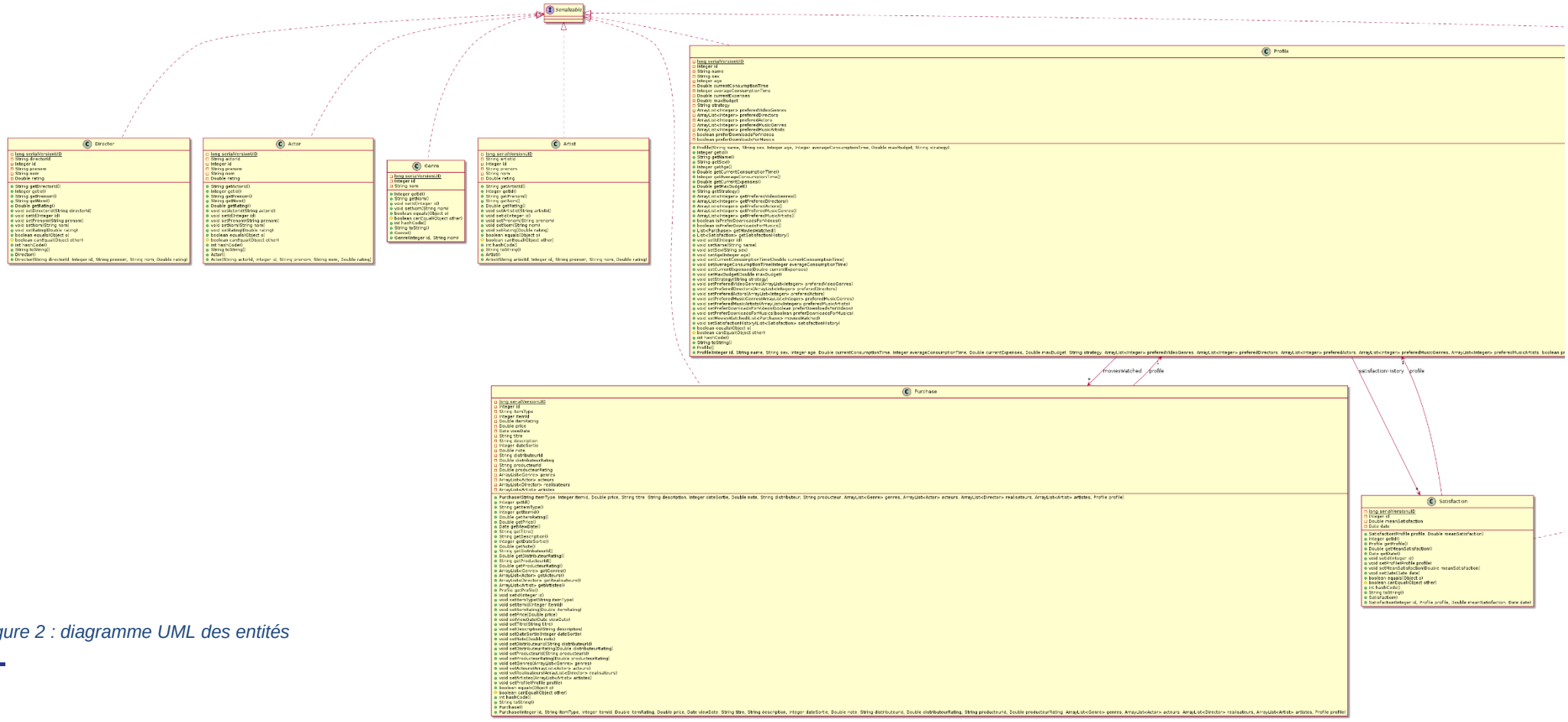


Figure 2 : diagramme UML des entités

Caractérisation des situations d’interaction

Nous n'avons pas réussi à échanger des ACL messages entre des agents qui n’étaient pas sur une même machine. Nous avons essayé sur un même réseau local d’une box WiFi, avec le même code, seuls deux de nos PC arrivaient à faire communiquer nos agents à distance… Nous avons cherché mais nous n’avons pas trouvé de solution à ce problème. Nous avons donc décidé de simuler les agents distributeurs et e-reputation sur notre container Jade au sein d’une même machine. Nous n’avons pas stimulé les producteurs car nous n'interagissons pas avec eux. Nous avons dans un premier temps voulu les simuler au niveau de test unitaire grâce à Mockito, mais finalement nous avons directement créé des Agent Jade dans le package mock. Ces Agents Mock utilisent des Behaviour Mock et renvoient des JSON Mock en réponse à nos ACL Message. Ces JSON Mock sont en accord avec les exemples que nous ont fournis les étudiants des groupes distributeurs.

Définition des comportements

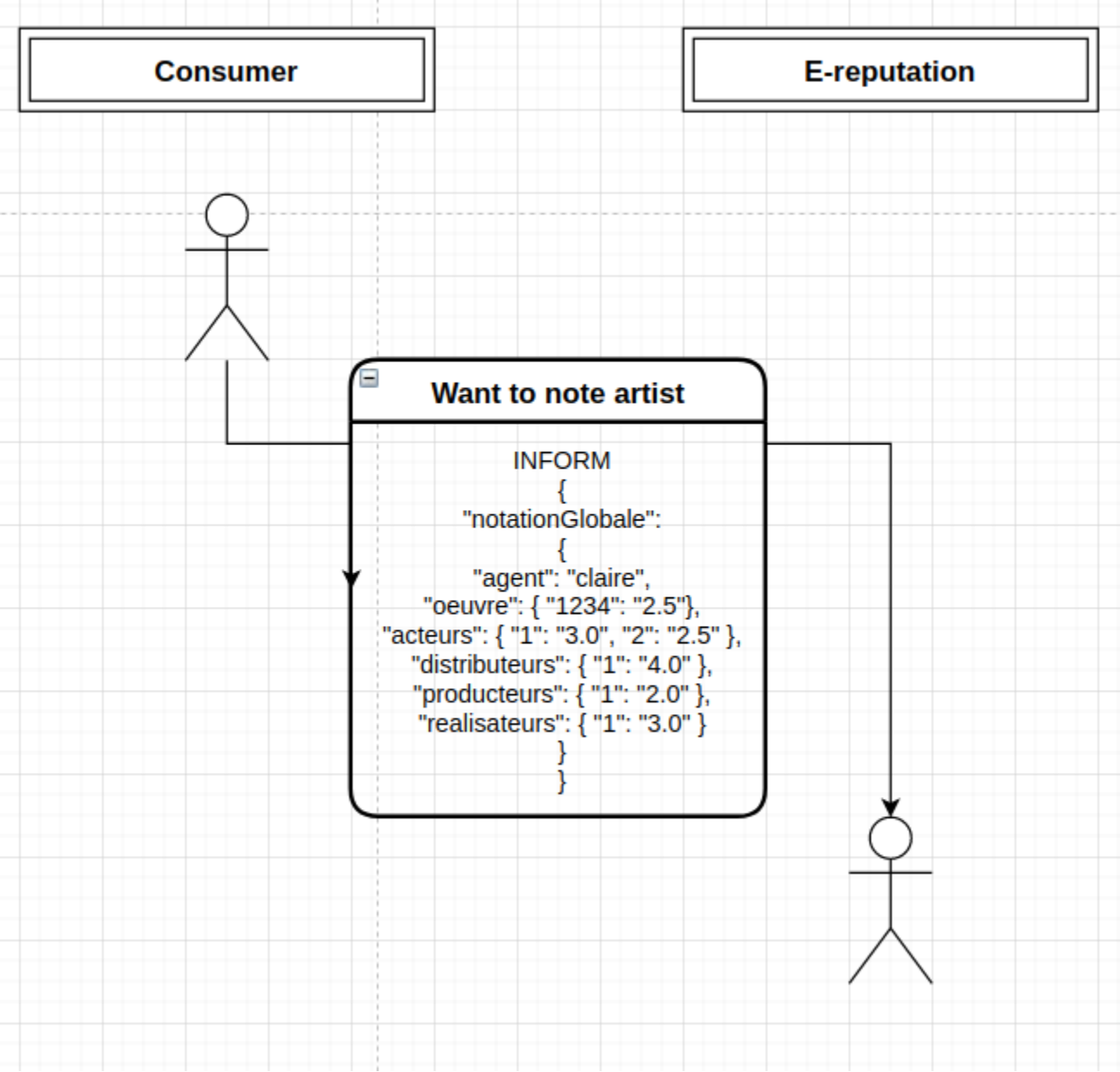


Figure 3 : échange

Communication entre l'interface graphique et l'agent internaute

Comme expliqué précédemment ...Elle est assurée par deux classes :

- la classe InternalComService du package service,
- la classe InternalComBehaviour du package "agent".

La première classe regroupe les méthodes chargées de façonner des messages JSON "stringifiés" qu'elles envoient ensuite à l'agent internaute cible via O2A, en les passant dans un objet Event.

La méthode chargée de cette dernière étape est la méthode sendToAgent qui prend en paramètre le type d'Event (code numérique défini par nous et permettant d'identifier la destination du message pour la suite), le message à envoyer, le nom de l'agent internaute cible (distributeurs ou e-reputation) et enfin un timeout. Cette méthode attend un retour de la part de l'agent destinataire dans la limite du timeout défini. Il le renvoie ensuite à l'API pour que celle-ci puisse répondre dans le body de la réponse HTTP (également sous forme d'un JSON "stringifié").

La méthode sentToAgent :

```
private String sendToAgent(int eventType, String jsonString, String agentName, int timeout) {
    AgentController agentController;
    try {
        agentController = JadeAgentContainer.getInstance().getAgentContainer().getAgent(agentName);
        Event event = new Event(eventType, this, jsonString);
        agentController.putO2AObject(event, AgentController.ASYNC);
        return (String) event.waitUntilProcessed(timeout * 1000);
    } catch (ControllerException | InterruptedException e) {
        System.err.println("Failed to send object to " + agentName + " : " + e.getMessage());
        return null;
    }
}
```

Côté agent, la classe InternalComBehaviour sert de "routeur". Cette classe est le seul Behaviour associé dans la méthode setup de notre agent. C'est un CyclicBehaviour dont le rôle est de réceptionner les Events envoyés par le service via O2A et de déclencher ensuite le bon Behaviour (OneShotBehaviour ou SequentialBehaviour) en fonction du type d'Event défini par le code numérique évoqué plus haut.

La méthode action de notre classe InternalComBehaviour :

```java
@Override
public void action() {
    // Recupération de l'event envoyé par le service
    Event event = (Event) myAgent.getO2AObject();

    // Event types
    // 0 -> Envoi des notes
    // 1 -> Recherche par titre
    // 2 -> Recherche par filtres
    // 3 -> Envoi acceptation de proposition

    // On dispatche à nos autres behaviours en fonction du type de l'event
    if (event != null) {
        if(event.getType() == 0) {
            myAgent.addBehaviour(new RateBehaviour(event));
        } else if(event.getType() == 1) {
            myAgent.addBehaviour(new SearchTitleBehaviour(event));
        } else if(event.getType() == 2) {
            myAgent.addBehaviour(new SearchFiltersBehaviour(event));
        } else if(event.getType() == 3) {
            myAgent.addBehaviour(new AcceptProposalBehaviour(event));
        } else {
            LOGGER.warning("Wrong event type was sent !");
        }
    } else {
        block();
    }
}
```

Chaque Behaviour de cette liste prend l'Event en paramètre de son constructeur, de façon à pouvoir renvoyer, grâce à la méthode "notifyProcessed", une réponse à ce dernier (et donc au final au service qui l'a créé) à la fin de ses traitements.

Interactions avec les autres agents

Format d'échange des messages

Nous avons décidé après une concertation entre responsables de groupes que la création d'une ontologie avec "Protégé" serait trop lourde à réaliser en comparaison de l'utilisation d'un système d'échange de messages au format JSON. En effet, il nous aurait fallu apprendre à utiliser un nouveau logiciel, alors que nous savions tous déjà créer et manipuler des objets JSON en Java.

Les messages au format JSON sont bien entendu transformés en "String" pour pouvoir passer dans la méthode **setContent** d'un **ACLMessage**. Lors de la réception par l'agent destinataire, celui-ci doit bien entendu réaliser l'opération inverse et recréer un objet JSON à partir de la chaîne de caractères.

A	B	C	D	E	F
Agent source	**Agent destination**	**Type de message**	**Exemple de message**	**Description**	**Message de retour**
all	E-réputation	INFORM	{ "notation": { "agent" :"Jacky", "type":"acteur", "id": "1000029", "note":"3.5" } }	Envoi d'une note pour un film || réalisateur || producteur || distributeur || acteur	{ "etat": "OK" || "KO", "id" : "10000029", "msg" : "Insertion de la note réussie" }
Internaute	E-Réputation	INFORM	{ "notationGlobale": { "agent": "claire", "oeuvre": { "1234": "2.5"}, "acteurs": { "1": "3.0", "2": "2.5" }, "distributeurs": { "1": "4.0" }, "producteurs": { "1": "2.0" }, "realisateurs": { "1": "3.0" } } }	Envoi d'une ou plusieurs notes - à noter que la note 0.0 n'est pas prise en compte mais que vous pouvez quand même le rentrer dans votre code.	{ "etat": "OK" || "KO", "msg" : "Insertion des notes réussie" }
all	E-Réputation	REQUEST	{ "popularite": { "type":"acteur", "id": "1000029" } }	Récupération de la popularité d'une oeuvre, d'un acteur, d'un réalisateur, d'un producteur ou d'un distributeur	{ "etat": "OK" || "KO", "id" : "1000029" "popularite": 0 < > 5 }
Producteurs	E-Réputation	REQUEST	{ "popularite ActeursListe": { "id": ["1", "2", "737", "4512"] } }	Récupération d'une liste de popularité pour des acteurs	{ "1": "4.0", "2": "3.5", "737":"2.0", "4512" : "Soit l'acteur n'existe pas, soit il n'a pas de notes." }
all	E-Réputation	REQUEST	{ "desirabilite": { "agent":"toto", "id": "1000029" } }	Récupération de la désirabilité d'une oeuvre pour un internaute donné (recommendation personnalisée en fonction de ce qu'il a déjà noté)	{ "etat": "OK" ou "KO", "id" : "1000029", "désirabilite": 0 < > 5 }

Figure 4 : Google Sheet avec l'ensemble des échanges

Stratégies

Stratégies définies pour notre agent

Le système de recommandation des œuvres proposées pour maximiser la satisfaction client est un des rôles des agents distributeurs. En effet, c'est à eux de recommander de manière optimale des œuvres en fonction du profil et de l'historique de l'internaute et ainsi d'essayer de générer le plus de profit. Dans notre cas, ce que nous appelons "stratégies" correspond plutôt à des profils type d'internautes. Nous avons défini quelques stratégies pour simuler le comportement de nos agents sur notre plateforme, nous avons donc :

- Econome,
- Exigeant,
- Streamer.

La stratégie **Économe** prend en considération la préférence de l'utilisateur pour le téléchargement ou l'abonnement, qui consiste à renvoyer à l'utilisateur la proposition la moins chère. Si l'utilisateur favorise le téléchargement, alors la stratégie compare uniquement le prix au téléchargement des œuvres proposées par les distributeurs et met en avant la moins chère. Sinon la stratégie compare le prix des œuvres et celui des abonnements proposés et met en avant l'offre la plus économe c.à.d la moins chère.

Ensuite, nous avons implémenté la deuxième stratégie, **Exigeant.** Cette stratégie devait initialement se baser sur des attributs des films tel que par exemple le type (.mp4, mkv etc) mais aussi sur la qualité (720p, 1080p, 4K etc). Or, les distributeurs n'ont pas implanté ce type d'indicateurs, nous avons donc dérivé et nous filtrons par date et par prix. Ainsi, la stratégie **Exigeant** permet de retourner à l'agent, l'œuvre la plus récente avec le prix le plus faible.

Quant à la dernière stratégie, **Streamer** est l'une des plus simples. Elle consiste à forcer la préférence abonnement et retourne seulement que l'offre d'abonnement la moins chère avec la plus longue durée.

L’argument prédominant entre la durée et le prix est la durée. Ainsi les offres qui se retournent, on aura par ordre croissant (de l’offre la moins chère à l’offre la plus chère).

Evaluation de la pertinence des stratégies

Ce que nous avons fait pour vérifier la pertinence de nos stratégies : nous évaluons tous les mois la satisfaction de nos trois différentes stratégies que nous comparons ensuite. Pour cela, nous avons une classe PassingTime dans le package commons, qui nous permet de simuler le temps qui passe. Pour nos tests nous avons utilisé la conversion suivante : 30 minutes <=> 30 jours. Ainsi, toutes les 30 minutes, c'est comme si un mois s'écoulait et on calcule la satisfaction moyenne du mois. Cette équivalence est paramétrable (s’il elle est trop lente pour l’analyse), on peut facilement accélérer ou ralentir, par exemple l'équivalence 1 seconde <==> jour.

À la fin du mois, pour calculer la satisfaction moyenne du mois, on appelle la méthode de la classe **Satisfaction**, qui calcule cette satisfaction moyenne à partir des ratios de temps consommé sur temps disponible et argent consommé sur argent disponible. En fonction des deux ratios, nous avons plusieurs échelles de satisfaction :

- 95% - 100% => très satisfait,
- 90% - 94% => Satisfait,
- 70% - 89% => Moyennement satisfait,
- 60% - 69% => peu satisfait,
- 0% -59% => pas satisfait.

On persiste en base chaque satisfaction moyenne et l'historique des satisfactions sont disponibles dans la vue /satisfaction. On peut simuler avec plusieurs profils type et voir qu’elles sont les stratégies qui apporte le plus de satisfaction moyenne.

Gestion de projet

Outils et technologies utilisés

Du fait de la situation sanitaire, il a été plus compliqué de communiquer, au sein d'un même groupe et entre groupes. Nous avons essentiellement utilisé Microsoft Teams et Discord pour pouvoir échanger à distance, car ce sont les outils que nous utilisons régulièrement. Au sein de notre groupe, nous avons également mis en place un tableau sur Trello afin de lister et répartir les tâches.

Nous avons tous utilisé l'IDE Eclipse et travaillé avec Maven pour la gestion des dépendances. La partie de l'application Java utilisée pour communiquer avec le frontend utilise Spring Boot. Le frontend est une application javascript codée avec VueJS.

La persistance des données nécessaires au fonctionnement de notre application est assurée par une base de données h2 locale. Nous communiquons également avec une base de données MariaDB hébergée sur alwaysdata.com pour récupérer certaines informations comme des listes d'artistes ou de genres musicaux pour la partie préférences de l'internaute. Cette base de données contient des informations sur les œuvres et a été mise en place par le groupe e-réputation (merci à eux).

Nous avons utilisé Git comme outil de versioning, via un dépôt sur Github.

Choix effectués

Le choix de VueJS pour la partie frontend et de Spring Boot pour la partie API de notre backend à été motivé par le fait que nous connaissions déjà ces technologies, vues en cours et/ou en entreprise. Nous aurions pu faire la partie graphique avec JavaFX ou faire une application web directement dans le projet Java avec un moteur de template. L'application web en javascript nous a semblé plus souple et plus rapide à coder. En revanche, le fait de séparer les applications frontend et backend nous a aussi obligé à mettre en place une API. L'avantage de cette solution est qu'il serait ensuite plus simple de créer une autre interface graphique (par exemple pour une application mobile) sans changer le reste du code.

Nous avons mis du temps à trouver comment nous allions faire communiquer notre API avec les agents Jade. Nous étions partis au départ avec l'idée de regarder du côté de JADE Web Service Integration Gateway (WSIG) mais son utilisation nous a parue peu claire. Nous avons finalement opté pour O2A pour faire passer des messages via des Jade Events entre nos services web et nos agents.

Difficultés rencontrées

La difficulté principale a été sans conteste le manque de temps. En effet, il ne nous a pas été possible de commencer à travailler sérieusement sur ce projet avant les dernières semaines précédant le rendu, d'autres projets pédagogiques à rendre à des dates antérieures nous prenant déjà tout notre temps. Notre groupe étant uniquement constitué d'alternants, nous n'avions que peu de temps libre à consacrer à ce projet pendant les périodes en entreprise.

Une autre difficulté a été le peu de documentation trouvée en ligne concernant certains points de l'utilisation de la plateforme Jade. Si la documentation sur l'utilisation de base est présente en abondance, des informations plus spécifiques, comme par exemple, la communication entre des classes "agents" et des classes Java "non agents", sont en revanche plus difficiles à trouver. Nous avons finalement déniché quelques exemples de l'utilisation de O2A sur Stackoverflow qui nous ont permis de mieux comprendre la Javadoc, mais cela a pris du temps.