Master 1 GIL - Document technique : Description de la bibliothèque de code

Groupe 1 Agora V3-1 : Partie bibliothèque de jeux

1er mai 2024

Version	6
Date	1er mai 2024
Rédigé par	MAZUY Axelle

Mises à jour du document

Version	Date	Modification réalisée
1	11 février 2024	Création du document
2	25 février 2024	Ajout de la partie bibliothèque des entités
3	3 avril 2024	Ajout des descriptions DTO Game et Player
4	24 avril 2024	Complétion des entités
5	29 avril 2024	Finition des entités Log et Message Explication sur les interfaces et générateurs Explication sur les services
6	1er mai 2024	Ajout des explicaitons contrôleurs et templates

Table des matières

1 Introduction **4**

2 Terminologie **4**

3 La bibliothèque des entités **5**
3.1 DTO Component 5
3.2 DTO Card 5
3.3 DTO Pawn 6
3.4 DTO Tile 6
3.5 DTO Token 6
3.6 DTO Game 7
3.7 DTO Player 7
3.8 Entité Help 7
3.9 Entité Log 8
3.10 Entité Message 8
3.11 L'interface GameParameters 8
3.12 L'interface GameTranslation 8

4 Les générateurs **9**
4.1 GameIdGenerator 9
4.2 PlayerIdGenerator 9

5 Les repositories **9**

6 Les services **10**
6.1 AbstractGameManagerService 10
6.2 GameManagerService 10
6.3 LogService 10
6.4 MessageService 11
6.5 PublishService 11

7 Les contrôleurs **12**
7.1 GameNavigationMenuController 12
7.2 MessageController 12
7.3 Architecture des contrôleurs 12

8 Les modèles de vue - templates **13**
8.1 Le fichier de chat 13
8.2 Le fichier de navigation 13
8.3 Le fichier de notification 13
8.4 Le fichier base 13
8.5 Architecture des templates d'un jeu 13

1 Introduction

Le projet Agora est une plateforme de jeux de société en ligne, permettant à des joueurs de partager un moment convivial à travers différents styles de jeux.

Pour la version V3-1 de l'application, deux équipes étaient en charge de la refonte :

— La première équipe était en charge de la partie bibliothèque de jeux (il s'agit du document actuel).

— La seconde équipe était en charge de la plateforme de l'application (un document du même style portant un nom similaire a été rédigé).

Vous trouverez donc ici la description du code de la bibliothèque de jeux. Au sein du code source de l'application, la partie jeux se trouve dans les dossiers nommés Game (permettant de distinguer facilement le code pour la plateforme et pour les jeux).

Il est également à noter qu'il n'y a de décrit ici que le code utilisable pour tous les jeux. Pour plus de détails sur les jeux implantés (6 qui prend, Splendor, Glenmore et Myrmes), référez vous aux DAL du jeu éponyme et au code source.

Enfin, il est à noter que la bibliothèque de code est vouée à être complétée et évoluer avec de nouveaux jeux.

2 Terminologie

3 La bibliothèque des entités

Il est d'abord à noter que les entités se distinguent en deux catégories : ce qui est trouvable en Base de données, et ce qui sert à la factorisation et l'abstraction des données (Data Transfer Obect - DTO).

Dans le cadre de la bibliothèque de code, nous nous pencherons d'abord sur les DTO et le rôle de chacune. Les DTO que nous étudierons se trouvent dans le dossier Entity/Game.

3.1 DTO Component

Cette classe permet de regrouper les caractéristiques communes aux différents composants de jeu que l'on peut rencontrer. Tous les composants ont donc un identifiant pour les reconnaître, mais aussi un attribut Help. Cet attribut permet d'associer à chaque composant de jeu une aide associée, pour expliquer notamment son rôle à l'utilisateur.

Dans le cadre d'un joueur novice, cela peut s'avérer particulièrement utile.

Help est une entité se trouvant en base de données (on peut retrouver cet objet dans le dossier Entity/Game).

L'intérêt premier de la classe Component est donc de regrouper toutes les caractéristiques communes à chaque composant de jeu. Toutefois, on peut subdiviser ces composants en différentes sous-catégories.

Contenu de la classe :

- id : l'identifiant du composant donné. En BDD permet d'identifier de manière unique un objet dans une table du composant spécifié
- help : une référence à l'aide apportée par rapport au composant, pour donner des indications au joueur. Ce champ est nullable car une aide peut ne pas être nécessaire.

3.2 DTO Card

De même que pour les composants, Card est la généralisation de toutes les cartes que l'on peut trouver dans les différents jeux implantés.

Cela ne signifie toutefois pas que les cartes ne peuvent pas avoir de spécificités en fonction des jeux. Si l'on souhaite définir un comportement précis à une carte, on peut alors définir une entité dédiée aux cartes d'un jeu spécifique dans le dossier du jeu concerné. Cette entité pourra alors étendre le comportement de Card.

Pour le moment, une carte générale n'a qu'une valeur. La signification de la valeur peut varier d'un jeu à l'autre, toutefois une carte se retrouve souvent avec une valeur (en points, un nombre significatif).

L'intérêt de la classe Card, tout comme pour Component, est donc d'être complété : soit en alimentant la bibliothèque d'attributs, soit en spécifiant un comportement unique à un jeu par la création d'un entité étendant l'objet Card.

Contenu de la classe :

- id (par héritage de Component) : l'identifiant du composant donné. En BDD permet d'identifier de manière unique un objet dans une table du composant spécifié
- help (par héritage de Component) : une référence à l'aide apportée par rapport au composant, pour donner des indications au joueur. Ce champ est nullable car une aide peut ne pas être nécessaire.
- value : la valeur d'une carte. Cette valeur dépend du jeu et peut être nullable si une carte ne possède pas de valeur.
- points : le nombre de points porté par une carte. Cela peut être le nombre de points que rapport cette carte au joueur et diffère de la valeur de celle-ci. Par exemple au 6 qui prend, la carte 21 vaut 1 point.

3.3 DTO Pawn

Tout comme les classes décrites précédemment, Pawn est la généralisation de ce que peut représenter, dans un ensemble de jeu, un pion.

Pour le moment, un pion est reconnaissable à sa couleur. En effet, dans beaucoup de jeu, un pion est notamment utile pour reconnaître le joueur sur un plateau.

L'intérêt de cette classe est donc une fois de plus d'être complété : un pion peut être reconnu par d'autres attributs, qu'ils soient ou non spécifiques à un jeu.

Contenu de la classe :

- id (par héritage de Component) : l'identifiant du composant donné. En BDD permet d'identifier de manière unique un objet dans une table du composant spécifié
- help (par héritage de Component) : une référence à l'aide apportée par rapport au composant, pour donner des indications au joueur
- color : une chaine de caractères indiquant la couleur du pion. Notamment utile pour différencier les joueurs.

3.4 DTO Tile

Une tuile peut être confondue dans certains jeux avec une carte. Toutefois, nous avons choisi de distinguer les tuiles et les cartes par le fait qu'une carte est unique, mais une tuile pas forcément. Une tuile a donc un type qui lui est associé.

Dans certains jeux, une tuile est une ressource de jeu qu'il faut récupérer. Elle possède donc un type qui permet de reconnaître la ressource qu'elle peut fournir.

Toutefois, cela ne suffit pas en soi, chaque jeu ayant sa notion particulière de tuiles et de cartes, il est important de ne pas seulement se poser la question de "qu'est-il écrit sur la notice ?" mais plutôt, dans le cadre de la bibliothèque de jeux, "quel but a cette ressource ?". Cela permettra ainsi de classifier entre carte et tuile, la nomenclature qui sera la plus appropriée.

Contenu de la classe :

- id (par héritage de Component) : l'identifiant du composant donné. En BDD permet d'identifier de manière unique un objet dans une table du composant spécifié
- help (par héritage de Component) : une référence à l'aide apportée par rapport au composant, pour donner des indications au joueur
- type : une tuile dans beaucoup de jeux ont un type différent comme dans Glenmore, où des tuiles peuvent être de type village, production etc.

3.5 DTO Token

Un jeton est également une ressource dans un bon nombre de jeux. Tout comme une tuile, il peut être considéré comme non unique, et permet de représenter un type de ressource spécifique.

C'est pourquoi un jeton a un type, permettant de classifier différents jetons d'un jeu.

Là encore, la liste des attributs n'est pas exhaustive. L'intérêt d'une bibliothèque étant d'être enrichie, si de nouveaux attributs viennent à être nécessaires, il ne faut pas hésiter à l'alimenter.

Contenu de la classe :

- id (par héritage de Component) : l'identifiant du composant donné. En BDD permet d'identifier de manière unique un objet dans une table du composant spécifié
- help (par héritage de Component) : une référence à l'aide apportée par rapport au composant, pour donner des indications au joueur
- type : une chaine de caractères indiquant le type de jetons, par exemple dans Splendor il existe plusieurs types de jeton : saphir, rubis, émeraude, etc.

3.6 DTO Game

Une partie de jeu possède des caractéristiques communes peu importe le jeu. Non seulement un identifiant, unique peu importe le jeu comme expliqué plus tard, ou encore un nom et des indicateurs de jeu.

Une partie a également d'autres caractéristiques communes mais il faut faire attention à l'héritage avec PHP et les entités.

Toutefois, comme pour les autres entités, la liste des attributs n'est pas exhaustive. Selon les évolutions, il est intéressant de compléter les caractéristiques d'une partie.

Contenu de la classe :

- id : l'identifiant d'une partie donnée. En BDD permet d'identifier de manière unique une partie peu importe le jeu
- launched : un booléen indiquant si la partie indiquée est lancée ou non
- paused : un booléen indiquant si la partie a été mise en pause, notamment après l'abandon ou l'exclusion d'un joueur, en attendant qu'il soit remplacé.
- gameName : une chaine de caractères comportant le nom du jeu de la partie.

3.7 DTO Player

Un joueur possède des caractéristiques communes peu importe le jeu. Non seulement un identifiant, unique peu importe la partie comme expliqué plus tard, ou encore un pseudo et des indicateurs de tour.

Un joueur est créé en fonction de la partie, un joueur est donc dépendant d'une partie et d'un utilisateur.

Un joueur a également d'autres caractéristiques communes mais il faut faire attention à l'héritage avec PHP et les entités.

Toutefois, comme pour les autres entités, la liste des attributs n'est pas exhaustive. Selon les évolutions, il est intéressant de compléter les caractéristiques d'un joueur.

Contenu de la classe :

- id : l'identifiant d'un joueur donnée. En BDD permet d'identifier de manière unique une partie peu importe le jeu
- turnOfPlayer : un booléen indiquant si le joueur doit jouer
- excluded : un booléen indiquant si a été exclu et doit être remplacé
- username : une chaine de caractères comportant le nom du joueur / son pseudo.
- score : un entier indiquant le score du joueur sur la partie en cours.

3.8 Entité Help

Cette entité sert notamment (comme abordé plus tôt) à donner des indications au joueur à propos d'un composant de jeu. Par exemple expliquer l'utilisation de cartes dans un jeu, etc.

Cette entité diffère des DTO précédentes notamment car elle est directement trouvable en BDD, il s'agit donc d'une table et non simplement d'un objet. C'est pourquoi une aide est référençable en tant que type dans un composant.

Contenu de la classe :

- id : l'identifiant d'une aide donnée. En BDD permet d'identifier de manière unique une aide peu importe le jeu
- gameName : une chaîne de caractères indiquant le jeu auquel l'aide est reliée.
- title : une chaîne de caractères donnant le titre de l'aide, notamment à quel composant l'aide est liée.
- description : une chaîne de caractères décrivant le comportement du composant et son utilisation au joueur.
- image : une chaîne de caractères contenant le chemin de l'illustration à laquelle l'aide est associée, pour aider notamment à la compréhension de l'utilisateur.

3.9 Entité Log

Cette entité sert notamment à enregistrer les actions, qu'elles soient utilisateurs ou système, en Base de Données. Elle peut avoir plusieurs utilités, et permet d'enregistrer à la fois une réussite, ou un échec.

Les logs peuvent notamment servir du côté plateforme pour voir l'inactivité d'un joueur.

Il peut également s'agir d'un bon indicateur de la santé de l'application.

Contenu de la classe :

— id : l'identifiant du log. En BDD permet d'identifier de manière unique un log peu importe le jeu.
— gameId : l'identifiant de la partie concernée par le log.
— playerId : l'identifiant du joueur ayant fait l'action. Peut être nul, notamment s'il s'agit d'un log système.
— message : une chaîne de caractères permettant de comprendre le motif du log.
— date : la date à laquelle le log a eu lieu.
— gameLabel : la chaîne de caractères permettant d'identifier le jeu concerné.
— time : l'heure à laquelle le log a eu lieu.

3.10 Entité Message

Cette entité sert notamment à enregistrer les messages envoyés sur les discussions entre joueurs, en Base de données. Les conversations sont publiques à tous les joueurs, mais ne sont pas accessibles aux spectateurs, notamment en écriture (auquel cas de la triche pourrait avoir lieu).

Contenu de la classe :

— id : l'identifiant du message. En BDD permet d'identifier de manière unique un message peu importe le jeu.
— content : le contenu du message, une chaîne de caractères en UTF-8.
— date : la date d'envoi du message.
— authorId : l'identifiant du joueur ayant envoyé le message.
— gameId : l'identifiant de la partie dans laquelle la discussion a lieu.
— authorUsername : le pseudo du joueur ayant envoyé le message.

3.11 L'interface GameParameters

Cette interface contient un ensemble de définition de constantes utiles pour l'ensemble des jeux. Il s'agit ici de constantes qui vont être utiles dans le cas de paramètres de fonction, ou de variables.

Pour l'heure, l'interface est séparée en deux groupes de constantes : les types de notifications, et les couleurs de notificiations.

Cette interface peut être facilement alimentée si des constantes utiles pour l'ensemble des jeux venaient à être nécessaires.

Il est également à noter que chaque jeu possède son interface <GameName>Parameters, permettant de regrouper de manière claire les constantes utilisées. Ces constantes sont regroupées par bloc sémantique, qu'il s'agisse du nom de ressources, de nombres spéciaux, etc.

De plus, les constantes déclarées en back dans ces interfaces, peuvent être importées en front grâce à twig par l'intermédiaire du mot-clé const ainsi que du chemin vers le fichier d'interface.

3.12 L'interface GameTranslation

Cette interface contient un ensemble de définitions de constantes utiles pour l'ensemble des jeux. Il s'agit ici de constantes qui vont être utiles dans le cas de messages, qu'il s'agisse de message de log, de réponses, d'affichage. Les constantes seront à la fois en anglais et en français (puisque le site est affiché en français aux utilisateurs, et les réponses sont en anglais).

4 Les générateurs

Comme cela a pu être expliqué dans la partie précédente, les joueurs et les parties de jeu sont uniques, quelque soit le jeu.

Cette unicité est définie grâce à un générateur.

4.1 GameIdGenerator

Cette classe est donc celle permettant d'avoir un identifiant unique à la création de chaque partie.

Dans cette classe, il n'est défini qu'une seule méthode : generateId. Cette méthode prend en paramètre un entity manager interface et une entité (un objet).

Il est à noter que les générateurs sont à compléter à l'ajout d'un nouveau jeu, étant donné que la fonction regarde dans chaque repository le plus grand identifiant enregistré pour l'incrémenter de 1.

4.2 PlayerIdGenerator

Cette classe est similaire en tout point au générateur d'id de partie. La seule différence est qu'elle regarde les repository de joueurs et non de parties de jeu.

Il faut donc également compléter ce générateur à chaque ajout de jeu.

5 Les repositories

Ces classes particulières sont utilisées pour interroger la Base de données. A chaque entité en Base de données correspond son fichier repository.

On y retrouve notamment des fonctions déjà créées et utilisables comme findOneBy, findAll, findBy et find, qui permettent à partir de critères donnés, de retourner les objets en BDD.

6 Les services

Les services correspondent à la partie code métier de l'application.
C'est donc ici que sont manipulées les entités et les repositories que nous avons pu voir précédemment.

6.1 AbstractGameManagerService

Cette classe abstraite est coupée en plusieurs parties :

— Dans un premier temps, on peut y voir des constantes, groupant ainsi le label de chaque jeu défini pour le moment, ainsi que des codes de retour (succès et erreurs).
— Dans un second temps, les signatures des méthodes liées à la gestion d'une partie de jeu : de sa création/suppression, à la création/suppression d'un joueur, ou à son lancement. Chaque méthode renvoie un code de retour correspondant aux constantes définies plus tôt.

6.2 GameManagerService

Ce composant permet de rassembler toutes les méthodes gérant une partie.
Il s'agit d'une classe appelant AbstractGameManagerService selon les paramètres qui lui sont donnés.
Il contient pour l'instant les méthodes suivantes :

Méthode	Type de retour	Description
createGame(string gameName)	int	Cette méthode permet de créer une partie du jeu identifiée par le nom gameName. Renvoie l'ID de la partie créée
joinGame(int gameId, User user)	int	Cette méthode permet de faire rejoindre l'utilisateur user dans la partie identifiée par gameId. Le joueur créé aura le même pseudo que son compte utilisateur. Renvoie un code de retour.
quitGame(int gameId, User user)	int	Cette méthode permet d'enlever user de la partie identifiée par gameId s'il s'agit d'un joueur de la partie. Renvoie un code de retour.
deleteGame(int gameId)	int	Cette méthode permet de supprimer la partie identifiée par l'entier gameId. Renvoie un code de retour.
launchGame(int gameId)	int	Cette méthode permet de lancer la partie identifiée par l'entier gameId si elle respecte les préconditions de lancement (nombre de joueurs dans la partie pour le jeu correct). Renvoie un code de retour.

6.3 LogService

Cette classe permet de manipuler l'écriture des logs dont on a défini le comportement dans les entités.
Comme nous avions pu le voir également, il existe différents types de log. Ce qui nous amène donc à avoir différentes méthodes pour leur écriture :

Méthode	Type de retour	Description
sendPlayerLog(Game game, Player player, String message)	void	Envoie en BDD le log du joueur indiqué avec le message donné
sendSystemLog(Game game, String message)	void	Envoie en BDD le log du système avec le message donné

6.4 MessageService

Cette classe permet de manipuler l'entité des messages dont on a défini le comportement dans la section des entités.

Les conversations entre joueurs sont disponibles dans chaque partie de jeux. Ce qui nous amène à avoir les méthodes suivantes :

Méthode	Type de retour	Description
sendMessage(int playerId, int gameId, string content, string authorUsername)	int	Crée un message et l'enregistre en BDD
receiveMessage(int gameId)	array	Renvoie le tableau des messages de la partie

6.5 PublishService

Cette classe permet de manipuler des changements réalisés sur une page web avec Mercure (sans avoir à recharger la page). Ce qui nous amène à avoir la méthode suivante :

Méthode	Type de retour	Description
publish(string route, Response data)	Response	Publie les données sur la route et met à jour la page web pour tous les écouteurs

7 Les contrôleurs

Les contrôleurs font le lien entre la vue et le modèle, respectant ainsi le modèle MVC.

Dans ces fichiers, on trouvera donc des "routes", des URL sur lesquelles les utilisateurs seront amenées en fonction de différentes actions sur la vue, qui auront un impact.

7.1 GameNavigationMenuController

Ce contrôleur permet de gérer les différentes actions liées au menu de navigation sur les jeux.

Méthode	Type de retour	Description
excludeAPlayer(int gameId, int playerId)	Response	Exclue le joueur d'identifiant donné de la partie donnée et met en attente la partie jusqu'à trouver un remplaçant
includeAPlayer(int gameId)	Response	Remplace un joueur exclue en changeant son pseudo et reprenant l'avancement du joueur exclu

7.2 MessageController

Ce contrôleur permet de gérer les conversations en jeu entre les différents joueurs de la partie.

Méthode	Type de retour	Description
sendMessage(int playerId, int gameId, string message, string authorUsername)	Response	Enregistre et publie le message aux autres joueurs
receiveMessage(int gameId)	Response	Récupère les messages de la conversation et les affiche visuellement

7.3 Architecture des contrôleurs

Il existe un contrôleur par jeu actuellement.

Une idée d'amélioration pourrait être, à terme, de créer des sous-dossiers par jeu et séparer les contrôleurs (qui comportent beaucoup de routes en fonction des jeux) afin de les rendre plus facilement parcourables.

D'une manière générale, les contrôleurs des jeux sont coupés en deux parties : les routes pour chaque action du jeu (souvent rangées par phase de jeu), et les publish (les méthodes qui permettront d'afficher via Mercure les changements effectués sur la page web en rechargeant les compsants visuels concernés).

8 Les modèles de vue - templates

Chaque jeu possède son dossier templates comme pour les entités et les services. Il est un peu moins visible qu'il existe une bibliothèque de code pour les fichiers de vue, toutefois certains fichiers restent communs aux jeux, et l'architecture est la même.

8.1 Le fichier de chat

Comme nous avons déjà pu aborder précédemment, il existe un chat de discussion entre joueurs dans les parties de jeu.

Ce chat est donc commun à tous les jeux, et il est donc compris dans le dossier Utils.

On retrouvera dedans de l'HTML, du Twig, du Tailwind et du JavaScript permettant d'ouvrir et fermer la fenêtre de chat, d'afficher les messages et d'en envoyer.

8.2 Le fichier de navigation

Sur chaque partie en cours figure également un menu de navigation, sous la forme d'un logo avec trois barres sur la partie supérieure gauche de l'écran. Ce fichier se trouve également dans le dossier Utils

Le code d'ouverture de ce menu figure donc ici, et les actions qui en découlent (comme l'exclusion d'un joueur) sont reliées au contrôleur GameNavigationMenuController que nous avons pu étudier précédemment.

8.3 Le fichier de notification

Enfin, il existe des notifications spéciales avertissant en tant réel le joueur des actions en cours (notamment qu'il s'agit de son tour de jouer, d'une erreur survenue, etc). Ce code est commun à tous les jeux, et se trouve donc dans ce fichier. Ce fichier est également dans le dossier Utils

8.4 Le fichier base

Nous allons maintenant aborder le sujet du fichier base. Dans un projet symfony, ce fichier n'est pas nécessairement voué à évoluer, et dans notre cas il s'agit d'un fichier qui a permis d'alléger les index des différents jeux. Comme son nom l'indique, ce fichier sert de base à étendre dans les autres fichiers de vue.

En utilisant les fonctions Twig, il est en effet possible d'étendre le comportement d'un fichier. Le fichier base étant généré automatiquement avec des blocks (des conteneurs Twig), il est possible de réécrire pour en changer le contenu (comparable au système d'override dans les méthodes).

La structure de base d'une page html avec les balises se trouve donc dans base, et il ne reste plus qu'à réécrire le contenu souhaité.

8.5 Architecture des templates d'un jeu

Concernant maintenant les jeux en eux-mêmes, nous avons respecté une architecture commune à tous : un dossier personalBoard, regroupant tous les fichiers utiles pour l'affichage visuel du plateau personnel dans les jeux, un dossier mainBoard faisant de même avec le plateau principal, un dossier ranking permettant de visualiser le classement et les informations générales en fonction des jeux, et on retrouve également le fichier index évoqué précédemment, ainsi que le fichier endGameScreenResult affichant le/les gagnant(s) de la partie (ou les perdants).