Master 1 GIL - Document technique : comment implanter un jeu de plateau sur Agora ?

Groupe 1 Agora V3-1 : Partie bibliothèque de jeux

3 mai 2024

Version	4
Date	3 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	Finalisation de la partie cahier des charges
3	3 avril 2024	Ajout de la partie architecture d'un jeu et des DTO
4	3 mai 2024	Finition de la documentation

Table des matières

1 Introduction

Lorsque l'on implante un jeu de plateau, il est important de définir un planning et une méthode précise. Il n'est pas nécessaire d'avoir déjà codé des jeux pour réussir à en implanter un. De plus, implanter un jeu de plateau et en développer un de zéro sont deux points très différents qu'il faut savoir prendre en compte.

Nous verrons donc dans ce document la démarche que nous avons suivi pour implanter les jeux qui nous ont été confiés (6 qui prend, Splendor, Glenmore et Myrmes).

Ce document comportera également un ensemble de questions que nous avons pu nous poser durant ce projet et qui aidera, je l'espère les futurs intervenants qui liront ce document.

2 Terminologie

DTO : ou Data Transfer Object (objet de transfert de données), permet comme son nom l'indique de transférer des données entre différents processus ou concepts logiciels. Ici on utilise cette notion pour factoriser les différents champs de nos entités en BDD afin de pouvoir facilement en rajouter/ en enlever si besoin. Nous pourrons donc utiliser dans les controllers et services communs les super classes afin de pouvoir factoriser des parties du code.

Mock : une imitation, désigne la méthode permettant de simuler le comportement d'autes objets.

3 Cahier des charges d'un jeu

Avant de commencer à coder, il est important de bien comprendre ce qui est demandé par le client, notamment pour un jeu.

On pourrait penser que le cahier des charges pour un jeu n'est défini que par sa notice, toutefois ce n'est pas complètement vrai et il est ainsi important de dialoguer avec le client pour voir ce qu'il aimerait vraiment.

Certains jeux ont des règles qui ne sont pas complètes, ou qui paraissent vagues et laissent un choix quant à leur interprétation. C'est dans ce genre de cas qu'il est essentiel d'instaurer un dialogue et dissiper les doutes.

En effet, même en connaissant le jeu, il est courant d'avoir une interprétation des règles différentes. C'est notamment ce qui a pu nous arriver avec le Splendor, où l'absence de remarque sur l'incapacité d'un joueur à mener son tour (par manque de possibilité d'action) a été un sujet de discussion avec le client.

3.1 Cas d'utilisation d'un jeu

Pour un jeu, la plupart du temps le seul cas d'utilisation est de jouer. Il peut toutefois arriver qu'un tour de jeu propose des actions diverses mais ne permettant d'en effectuer qu'une seule, ou une action qui peut s'effectuer à tout moment.

Dans ce cas, le diagramme des cas d'utilisation prévoit d'autres cas d'utilisation correspondants aux actions parallèles.

Dans le cas de Splendor par exemple, les 4 actions possibles ne peuvent s'effectuer en même temps. Il faut donc étendre le cas d'utilisation "jouer un tour" pour définir ce que cela peut signifier concrètement (acheter une carte, réserver une carte, prendre deux jetons de la même couleur ou 3 jetons de couleurs différentes). De même pour Glenmore, l'action de vente des ressources peut s'effectuer n'importe quand en parallèle des autres actions du tour, toutefois, cette action n'est possible que si c'est le tour du joueur, on étend donc le comportement du cas "jouer un tour" pour le compléter par "vend ses ressources".

3.2 Exigences fonctionnelles

Les exigences fonctionnelles d'un jeu correspondent aux règles de gestion concernant les règles du jeu.

Il est important de les formuler de manière à pouvoir les tester dans le cahier de recettes par la suite.

Elles doivent donc être testables, mesurables et complètes. Il est normal d'avoir beaucoup d'exigences fonctionnelles en fonction des jeux. Un jeu plus ou moins complexe peut avoir plus d'une cinquantaine d'exigences fonctionnelles.

Pour établir la liste des règles d'exigences fonctionnelles, il est utile d'établir une liste des actions possibles pour un joueur lors d'une partie. A chaque évènement, que peut faire le joueur ? Prévoir tous les cas possibles permet également de pallier à des abus de comportement.

4 Architecture d'un jeu

4.1 Rédaction du Document d'Architecture Logiciel

Le Document d'Architecture Logiciel constituera la feuille de route du jeu à implanter. Plusieurs éléments y sont essentiels pour faciliter le développement qui suivra.

L'un des plus grands conseils pour la rédaction de ce document, est de faire des réunions récurrentes. Confronter les idées est essentiel quand il s'agit d'architecture, et un seul esprit ne suffit pas à avoir assez de recul. Pour notre part, nous avons tenu 2 réunion par semaine de plusieurs heures, durant lesquelles nous discutions et mettions en commun notre avancée tout en nous concentrant point par point sur l'architecture. Cela s'est révélé être plus efficace que le début de notre projet qui était fait avec des réunions à distance via le canal de discussion.

Il est également essentiel que tous les membres s'intéressent à l'architecture des jeux, sans quoi le développement se révèlera complexe et pourra relever de quelques personnes uniquement, ce qui constitue une charge très lourde à porter en plus de se révéler dangereux pour l'avancée du projet.

Afin de réaliser l'architecture d'un jeu, il faut donc en connaître parfaitement les règles et reprendre le cahier des charges réalisé.

4.2 Le MCD

Le MCD est donc le modèle de données du jeu. Etant donné qu'Agora fonctionne avec Symfony, créer une entité revient à avoir une table associée en Base de Données (ce qui, dans d'autres langages, pourrait être une charge de travail supplémentaires à réaliser pour relier les deux).
Pour réaliser le MCD, il faudra donc analyser l'existant, notamment avec les composants se trouvent actuellement dans la bibliothèque de code (voir section suivante sur le développement d'un jeu).

Ce qui s'est révélé pour nous être le plus efficace a été de séparer le jeu en différentes zones puis en différents composants : par exemple, dans le 6 qui prend, il faut gérer un plateau de 4 lignes de cartes. Il faut donc une entité correspondant à une ligne, mais aussi une entité correspondant au plateau principal, et à une carte.

Il n'est toutefois pas anormal de devoir modifier le MCD en cours de développement. Attention à ne pas tomber dans le piège de l'analyse à ne plus toucher. Même une fois le DAL réalisé et avec une très bonne analyse, il est normal de devoir remettre en question de temps à autre ce qui avait pu être écrit. C'est en développant que l'on se rend parfois compte d'une réalisation impossible ou trop coûteuse.

4.3 La séparation en services et méthodes

Après avoir créé votre MCD, il faudra vous occuper du code métier, autrement dit des services.

Information : si vous utilisez SonarQube, celui-ci vous donnera un avertissement si vos classes dépassent les 20 méthodes, veillez à bien séparer vos services.

Pour séparer vos services, il peut être intéressant de le faire par notion de jeu. Par exemple, dans Splendor, la gestion des jetons est une notion à part entière, il convient donc de lui dédier un service. Cela vous permettra de respecter l'un des principes SOLID : SRP (Single Responsability Principle), qui voudrait qu'à une classe on associe une seule responsabilité (on ne mélange donc pas les notions).

4.4 Les contrôleurs

La dernière étape est donc de construire les différentes routes dont vous pensez avoir besoin pour votre jeu.

De notre point de vue, il s'agissait de la partie la plus techniquement compliquée à concevoir en amont. En effet, elle dépend de la vue, des boutons et actions que l'on y fera, mais aussi de ce qui aura été réalisé en code métier.

Pour nous, le plus simple a été de simuler une partie de jeux, et pour chaque action que l'on réalisé (que l'on déplace quelque chose, que l'on prenne un objet), une route était associée.

L'élaboration des contrôleurs est donc sûrement celle qui est la plus vouée à évolution durant votre développement.

4.5 Les diagrammes de séquence

Votre DAL sera sûrement séparée en deux types d'architecture si vous vous fiez aux DAL des années passées : architecture statique et architecture dynamique. De notre côté, l'architecture dynamique désignait donc l'interaction entre nos composants pour expliquer le scénario qui se produira à chaque action.

A cet effet, des diagrammes de séquence sont tout indiqués pour décrire les appels de méthodes, les paramètres, etc qui auront lieu. Toutefois cette tâche peut s'avérer chronophage, notamment sur des gros jeux comme Myrmes, avec beaucoup de routes. Ne vous perdez donc pas dans les diagrammes de séquence.

Toutefois, pour ceux que vous aurez, cela pourra s'avérer être un précieux outil, notamment pour faire comprendre à l'ensemble des membres du groupe la communication entre les différents composants.

5 Développement d'un jeu et intégration à l'existant

5.1 La bibliothèque de code

Il est tout d'abord essentiel de comprendre la bibliothèque de code mise en place. Cette bibliothèque est notamment utile pour la mise en commun des caractéristiques communes à un ensemble de jeux.

5.2 La bibliothèque d'entités

Les entités peuvent être factorisées avec des caractéristiques communes, comme la valeur d'une carte par exemple.

Ces caractéristiques identifiées peuvent être complétées. Il n'est pas nécessaire que tous les jeux soient concernés, un champ non utilisé vaut mieux qu'un champ dupliqué dans plusieurs jeux.

Pour utiliser cette bibliothèque, il faut, à la création des entités d'un nouveau jeu, étendre la DTO concernée.
Par exemple, si un nouveau jeu à implanter utilise des cartes, notons ce jeu avec le sigle NJ, on pourra créer une entité CardNJ. Cette entité étendra Card et récupèrera ainsi les champs qui y sont définis afin de les mettre en Base de données. En effet, les DTO n'ont rien en Base de données, ce sont les entités en héritant qui le seront.

Il existe plusieurs niveaux de factorisation dans la bibliothèque afin de compléter le plus possible le comportement des différents composants d'un jeu. Par exemple : Component désigne un composant d'un jeu, pouvant être une Card (carte), un Token (jeton), un Pawn (pion). On peut retrouver également une DTO Player désignant un joueur et les caractéristiques communes peu importe le jeu, ou encore Game pour une partie de jeu.

Pour avoir une description détaillée de chaque composant actuellement dans la bibliothèque de code, vous pouvez vous référer à la documentation à ce propos.

Pour créer une entité, vous pouvez utiliser dans le conteneur agora la commande suivante : symfony console make :entity
Le reste de la déclaration de votre entité est alors interactive. Une fois votre entité créée, si vous voulez utiliser un composant de la bibliothèque de code, vous pouvez alors simplement faire étendre votre entité avec le composant souhaité, vous récupèrerez ainsi les attributs de ce composant dans votre entité (et donc dans votre table en BDD).

5.3 La complétion des Repositories

Un autre point lié aux entités, est la définition des repositories (les classes permettant d'interroger la Base de données sur nos entités).
Lorsque vous créez des entités en utilisant le make :entity de Symfony, il est également généré une classe repository associée.
Il vous est possible d'y définir vous-même des méthodes de récupération particulières, ou vous pouvez utiliser les méthodes définies par défaut dans les repositories.

5.4 Mettre à jour les générateurs

Si vous avez lu préalablement la description de la bibliothèque de code sur les jeux, vous avez dû rencontrer la notion de générateur.
Il est donc essentiel, lors de l'ajout d'un jeu, de mettre à jour ces générateurs en y ajoutant la vérification sur le jeu ajouté.
Sur la capture d'écran ci-jointe, vous pouvez voir que 3 lignes se répètent à chaque jeu, seul change le repository associé. Il faut donc effectuer la même chose avec le jeu implanté.

```php
public function generateId(EntityManagerInterface $em, $entity) : mixed
{
    $game = $this->gameSixQPRepository->findOneBy([], ['id' => 'DESC']);
    $id = $game == null ? 0 : $game->getId();
    $lastId = $id;

    $game = $this->gameSPLRepository->findOneBy([], ['id' => 'DESC']);
    $id = $game == null ? 0 : $game->getId();
    $lastId = max($id, $lastId);

    $game = $this->gameGLMRepository->findOneBy([], ['id' => 'DESC']);
    $id = $game == null ? 0 : $game->getId();
    $lastId = max($id, $lastId);

    $game = $this->gameMYRRepository->findOneBy([], ['id' => 'DESC']);
    $id = $game == null ? 0 : $game->getId();
    $lastId = max($id, $lastId);

    return $lastId + 1;
}
```

FIGURE 1 – Exemple de générateur à compléter

Cela permet de vérifier dans toutes les tables des jeux le plus grand identifiant enregistré afin de ne pas avoir de conflit avec 2 identifiants identiques pour deux jeux différents.

Il faut donc mettre à jour le générateur d'identifiant de partie, mais aussi de joueur (qui est un miroir de celui de game).

5.5 Les migrations et fixtures

Si vous avez regardé le code existant, vous avez pu voir que les données de jeu (comme les cartes du 6 qui prend , les tuiles du Glenmore etc) sont dans des fichiers dans le dossier migrations. Les migrations sont des lignes de commande qui sont entrées en BDD, toutefois, les données de jeu devraient se trouver dans des fichiers fixtures (comme nous avons pu en discuter avec le référent technique).

Les fichiers de fixtures se présente comme des méthodes en php, où vous écrivez de la même manière que dans vos services la déclaration et l'initialisation de vos entités, que vous allez persist (enregistrer en Base de données) puis flush (mettre à jour la Base de données) grâce à un ObjectManager.

Cela peut s'avérer très long et rébarbatif, toutefois, cela constitue une étape incontournable pour l'implatation de votre jeu.

5.6 Les services

Maintenant, concentrons nous sur la partie code métier de votre jeu. Vous en avez fait l'analyse et savez les méthodes que vous allez utiliser, toutefois, avant de vous lancer dans le développement de ces classes, il vous faudra créer une classe <GameName>GameManagerService.

Cette classe étendra le comportement de AbstractGameManagerService qui est définie dans la bibliothèque de code. C'est donc cette classe que vous allez implémenter qui va vous permettre de construire une partie de votre jeu, créer les joueurs, les supprimer, supprimer une partie, la lancer.

Vous pouvez vous inspirer de ce qui a été précédemment fait, notamment dans le SixQPGameManagerService, pour comprendre comment écrire votre code.

Toutefois, il s'agit ici d'une initialisation de base des différents éléments. Pour une initialisation plus poussée, créez une méthode initialize dans l'un de vos services.

Attention, pensez également à compléter la classe GameManagerService avec le jeu que vous implantez.

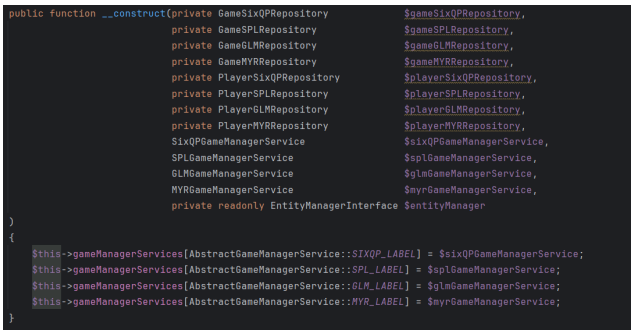

FIGURE 2 – Constructeur du GameManagerService

Vous devrez tout d'abord ajouter ici votre <GameName>GameManagerService, ainsi que votre repository pour le joueur et la partie. Vous pouvez également voir que vous devrez ajouter un label avec votre nom de jeu dans AbstractGameManagerService, et vous pourrez initialiser ici votre tableau.

```php
public function getGameFromId(int $gameId): ?Game {
    $game = $this->gameSixQPRepository->findOneBy(['id' => $gameId]);
    if ($game == null) {
        $game = $this->gameSPLRepository->findOneBy(['id' => $gameId]);
    }
    if ($game == null) {
        $game = $this->gameGLMRepository->findOneBy(['id' => $gameId]);
    }
    if ($game == null) {
        $game = $this->gameMYRRepository->findOneBy(['id' => $gameId]);
    }
    return $game;
}
```

FIGURE 3 – getGameFromId

De même vous allez voir 3 méthodes nécessitant également que vous les complétiez :
— **getGameFromId**
— **getPlayerFromid**
— **getExcludedPlayerFromGameId**
Ces méthodes sont à compléter avec les repositories correspondants pour votre jeu.

L'ensemble des méthodes à modifier a été simplifié au maximum pour qu'il n'y ait pas tant de modifications à apporter aux futurs intervenants, toutefois il en subsiste quelques unes comme vous pouvez le voir.

5.7 Les contrôleurs

Vous allez maintenant faire vos contrôleurs. Tout d'abord, il est à noter que les contrôleurs étendent AbstractController.

Comme toutes les classes, ils possèdent un constructeur, qui contiendra notamment toutes les classes que vous utilisez.

Mais passons maintenant à la méthode showGame. Cette route sera celle avec l'URL la plus basique pour votre jeu. Essayez de respecter la syntaxe suivante :
'/game/<gameName>/{id}' avec id l'identifiant de la partie.

De même, quelques lignes se répèteront quelque soit la route que vous allez développer.

Prenons l'exemple du 6 qui prend pour illuster cela.

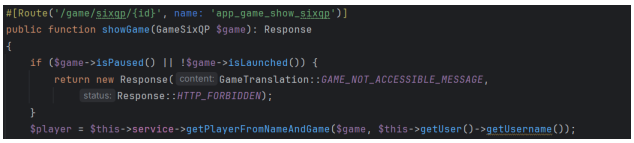

FIGURE 4 – Début du showGame du jeu 6 qui prend

Comme on a pu le voir précédemment, la route possède une URL simple, mais aussi un name. Ce nom sera celui utilisé par le front pour appeler les routes (cela permet ainsi de ne pas faire dépendre des paramètres, et rend plus maintenable les routes).

De plus, on peut voir qu'une vérification de la pause du jeu ou du lancement du jeu est réalisée : cela est dû à la gestion de l'exclusion d'un joueur qui met en pause la partie jusqu'à son retour. On empêche donc les joueurs de visualiser ou jouer leurs actions.

Un autre point concernant les contrôleurs consistent en l'enregistrement des logs. Il ne faut pas oublier d'appeler le service de logs pour enregistrer les coups effectués dans la partie, ou par le système. De même, il faut envoyer des réponses adaptées (se renseigner sur les codes de retour HTTP).

Enfin, parlons des render. Les render servent à renvoyer un fichier de vue précis, avec des paramètres Twig. C'est ce qui permet de faire de la communication entre front et back. C'est ici que l'on peut passer des informations des méthodes pour adapter la vue.

5.8 Les tests automatiques

5.8.1 Les tests unitaires

C'est ici que sont réalisés les tests des entités et services, mais pas des contrôleurs.

Un test unitaire permet de tester l'unité la plus petite possible du code. Autrement dit, vous testez par exemple un cas de succès sur votre méthode, ou un cas d'échec. Veillez à bien couvrir l'ensemble des branches de votre méthodes.

Pour suivre la progression de la couverture du test sur votre code, vous pouvez écrire la commande suivante : XDEBUG_MODE=coverage php bin/phpunit -d memory_limit=-1 –coverage-html reports/
Cette commande va générer des rapports html (voir le dossier reports, puis ouvrir index.html sur votre navigateur) vous indiquant quelles lignes sont couvertes par vos tests. Le memory_limit=-1 est présent pour ne pas être bloqué par la mémoire prise par les tests (étant donné qu'il y en a un nombre conséquent).

En test unitaire, on n'utilise pas la Base de données, on utilisera donc des Mocks (ou bouchons) pour simuler le comportement de certaines fonctions.

5.8.2 Les tests d'intégration

Similaires aux tests unitaires sur leur portée, les tests d'intégration utilisent quant à eux la Base de données et les éléments extérieurs (comme les API). On n'utilise donc plus de Mocks, mais des données réelles, afin de tester le comportement des méthodes en contexte.

5.8.3 Les tests d'application

Les tests d'application portent quant à eux sur le contrôleur. Ils permettent de tester les routes du contrôleur et leur action sur le code métier.

Il s'agit donc du test automatique permettant de vérifier l'ensemble d'une fonctionnalité.

Le test d'application va simuler un client utilisant l'application.

5.8.4 Adaptation des tests de la factory

Comme on a pu l'aborder précédemment, lorsque vous implantez votre jeu de plateau, vous allez devoir ajouter des attributs dans GameManagerService, et modifier certaines méthodes.

Il faudra donc en conséquence adapter les tests pour rajouter dans le setUp l'initialisation nécessaire.

FIGURE 5 – Méthode setUp du service de test unitaire du GameManagerService

Après avoir mis à jour le setUp, il vous faudra également ajouter quelques lignes dans les 2 méthodes qui suivent :

— createGameManagerServiceWithMockFunctionRepository
— createGameManagerServiceWithMockFunctionWillReturn

Vous devrez ajouter les mêmes lignes que pour les autres jeux, mais avec votre jeu.

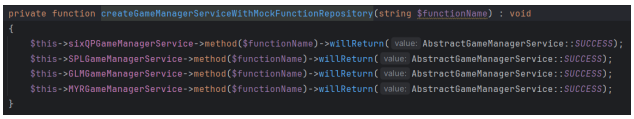

FIGURE 6 – createGameManagerServiceWithMockFunctionRepository

FIGURE 7 – createGameManagerServiceWithMockFunctionWillReturn

5.9 Les stimulus contrôleurs et fichiers Javascript

5.9.1 Les animations

Les animations constituent un point important de la vie d'un jeu. Bien qu'il ne s'agisse que d'un visuel qui ne met pas en péril les données du jeu, il s'agit d'un élément permettant une meilleur compréhension pour l'utilisateur.

Qu'elles fassent office de transition, ou permettent seulement d'avoir les informations affichées directement aux yeux de l'utilisateur, les animations sont un vaste sujet sur lequel il faut pouvoir se pencher.

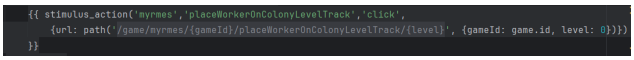

FIGURE 8 – Portion de code du fichier templates displayColonyLevelTrack.html.twig du jeu Myrmes, bouton plaçant une fourmi sur un niveau de fourmilière

Attention à ne pas prévoir trop d'animations, au risque de perdre du temps et surcharger l'utilisateur d'informations.

Pour savoir comment implanter une animation, se référer à la documentation concernée.

5.9.2 Les stimulus contrôleurs

Si vous avez déjà fait des projets reliant front et back, vous avez sûrement déjà géré les actions de boutons par exemple.

En utilisant Twig, il est possible de faire appel à Stimulus. Cela peut se manifester de la manière suivante :

Vous pouvez voir ici une portion du code du bouton permettant de placer une ouvrière au niveau 0 de fourmilière sur Myrmes.

Vous pouvez donc ici voir ce qu'il se passe quand on clique sur le bouton : on appelle une fonction stiumulus_action, prenant en paramètre plusieurs éléments :

- Le premier paramètre est le nom du fichier stimulus. Si vous allez voir dans assets, controllers, vous verrez des fichiers comme myrmes_controller.js, le paramètre prend donc en compte la première partie du nom du fichier (le préfixe avant controller). Cela permet donc de faire appel à ce fichier pour la suite.
- Le deuxième paramètre est le nom de la méthode à appeler. Elle doit donc faire partie du fichier appelé avec le premier paramètre.
- Le dernier paramètre constitue le paramètre pris par la méthode donnée précédemment. Ici, le paramètre demandé est une URL vers la route du contrôleur PHP, afin d'exécuter l'action en back. On utilise donc la méthode url :path, prenant le nom de la route du contrôleur PHP voulue, et donnant les paramètres de cette route.

Bien que pouvant sembler un peu complexe au premier abord, cela deviendra un automatisme à mesure du développement. Les fichiers stimulus sont des fichiers écrits en JavaScript, n'ayant pas d'autre particularité que de relier un élément à une action. La plupart du temps, les routes du contrôleur Stimulus se contentent d'appeler les routes en back avec un fetch.