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

Groupe 1 Agora V3-1 : Partie plateforme Web

19 Mai 2024

Version	9
Date	19 Mai 2024

Rédigé par :
Robin SAPIN
Yacine BEN AHMED
Sid Ahmed BRAHIMI
Corentin PILLE
Mohamed CHERFI
Michel NASSALANG
Anass EL GHARBAOUI

Mises à jour du document

Version	Date	Modification réalisée
1	8 Avril 2024	Création du document
2	15 Avril 2024	Ajout de la partie Controlleur
3	22 Avril 2024	Ajout partie templates
4	23 Avril 2024	ajout templates
5	28 Avril 2024	templates home, dashboard, registration et security
6	2 Mai 2024	Ajout des entités
7	10 Mai 2024	Services BoardManager, GameViewer
8	13 Mai 2024	Services Notification, User
9	15 Mai 2024	Ajout de fonctions dans le BoardControlleur et le BoardManagerService
9	19 Mai 2024	templates admin

Table des matières

1 Introduction

Le but de ce document est de fournir une description technique et exhaustif de l'ensemble du code qui compose la partie interface web du projet Agora, afin de permettre, lors d'une itération ultérieure du projet pour une année suivante, de reprendre facilement le code existant pour le groupe chargé de la nouvelle itération. Une description précise et complète est nécessaire afin de permettre aux futurs groupe de comprendre aisément le fonctionnement des entités, contrôleurs, services et templates développés, pour permettre des modifications et des ajouts de fonctionnalités au projet, mais aussi de garantir sa scalabilité.

L'architecture de ce document suit le même format que l'architecture logicielle du projet, cela permettant au lecteur de pouvoir retrouver facilement la partie du code qui l'intéresse. Il est aussi à noter que le but de ce document n'est pas de fournir des guides précis (par exemple "Comment ajouter un jeu à la plateforme ?"), mais seulement de décrire le but de chaque fichier qui compose le projet. La réalisation de ce type de guide devra donc se faire sur des documents à part de celui-ci.

2 App

2.1 src

2.1.1 Controller/Platform

2.1.1.1 AdminController

2.1.1.2 BoardController

Ce contrôleur s'occupe de la redirection de toutes les pages et intéractions avec comme cible principal un Board. Ses attributus (initialisés dans son constructeur) sont les suivants :

- EntityManagerInterface $entityManagerInterface : L'interface permettant notamment de récupérer les différentes entités utilisées par le controlleur via leur id
- BoardManagerService : Le service utilisé pour les différents appels au code métier réalisé par le contrôleur, agissant sur un Board
- GameViewerService : Le service utilisé pour les différents appels au code métier réalisé par le contrôleur, agissant sur la vue d'une partie
- Security $security : L'interface utilisé notamment pour récupérer l'entité User de l'utilisateur courant via son id.

create(Request, Int) :
Route : /dashboard/boardCreation/game_id

Cette fonction permet de genérer un formulaire de création de Board, associé au Game d'id $game_id qui est passé en paramètre via la route de ce contrôleur.
Il va d'abord créer la nouvelle entité Board, puis générer le formulaire de type BoardRegistrationType, en passant l'id du Game via le paramètre $game dans la fonction createForm.

Une fois le formulaire complété par le joueur, le contrôleur fait appel au service BoardManagerService afin de correctement initialisé le Board selon les paramètres renseignés par le joueur, ainsi que d'ajouter ce joueur à la table (via l'appel à addUserToBoard). Le contrôleur ne procède à aucune vérification des données envoyées, puisque ceux-ci sont contrôlées directement dans le front du formulaire.

Une fois le formulaire validé, le controlleur redirige le joueur sur la page 'app_dashboard_user', donc la page d'accueil de la dashboard du joueur, avec sa nouvelle table d'affichée et rejointe, avec un message flash confirmant l'action (message modifiable dans l'appel à la fonction addFlash.

Il est à noter que comme la création d'une table peut seulement se faire si l'utilisateur est un joueur, la route utilisée a donc pour racine /dashboard.

joinBoardController(Int) :
Route : /joinBoard/id

L'utilisation de cette route doit être faite lorsque l'on souhaite inscrire le joueur ayant cliqué sur le lien à un Board.

La fonction va d'abord récupérer l'entité Board concerné, via son id passé en paramètre de la fonction depuis la route, puis récupérer l'entité User associé au joueur ayant cliqué sur le lien, via l'interface Security. On vérifie ensuite que la table est valide pour y inscrire le joueur, en vérifiant la disponibilité de la table.

En cas de succès, on inscrit le joueur à la table via le service BoardManagerService, puis on le redirige sur sa page d'accueil de dashboard 'app_dashboard_user', avec un message flash confirmant son action . En cas d'échec, on fait un appel à la route 'app_boards_game' avec en paramètre l'id du Game du Board que

le joueur a tenté de rejoindre, avec un message flash indiquant l'erreur (contenu dans $errorMessage).

leaveBoard(Int) :
Route : /leaveBoard/id

L'utilisation de cette route doit être faite lorsque l'on souhaite retiré le joueur ayant cliqué sur le lien d'un Board.

La fonction va d'abord récupérer l'entité Board concerné, via son id passé en paramètre de la fonction depuis la route, puis récupérer l'entité User associé au joueur ayant cliqué sur le lien, via l'interface Security.

On réalise enfin un appel à la fonction removePlayerFromBoard du service BoardManagerService afin de retirer correctement le joueur du Board, puis on le redirige sur la page Tables de la dashboard.

showGame(Int) :
Route : /showGame/id

L'utilisation de cette route doit être faite lorsque l'un joueur souhaite afficher la vue d'une partie.

La fonction va d'abord récupérer l'entité Board concerné, via son id passé en paramètre de la fonction depuis la route, puis récupérer le label de ce Board afin de l'utiliser comme paramètre de la fonction getGameViewRouteFromLabel du service GameViewerService pour récupérer la route de la vue associé au Game dont la partie que le joueur souhaite voir en est une instance. Elle fait à la fin la redirection vers cette route en passant en paramètre l'id de la partie du board.

Comme c'est la route retournée par le GameViewerService qui s'occupe de faire la distinction entre un joueur est un spectateur, la route /showGame peut donc être utilisé, peut importe si c'est pour joueur sa partie ou en regarder une.

checkInvitation(Notification, Int) :
Route : /checkInvitation/notification_id/id

L'utilisation de cette route doit être faite lorsqu'un joueur rejoint une table auquelle il a été invité, en cliquant sur la notification d'invitation

La fonction va d'abord récupérer définir la notifcation d'invitation sur laquelle le joueur à cliquée (désignée par notification_id) comme lue (pour éviter de la revoir après un premier clic). Elle va ensuite récupérée l'entité Board concernée, via son id passé en paramètre de la fonction depuis la route, puis vérifier si le joueur est bien présent dans la liste des invités de la table, ainsi que vérifier que la date (jour + heure) de son clic n'a pas dépassé celle du timer d'invitation (autrement dit, on vérifie que l'invitation n'ait pas expiré).
Si cette vérification est positive, le controlleur redirige vers le controlleur *joinBoard*, afin de faire entrer le joueur dans la table. Par contre si celle-ci est négative, le joueur est redirigé vers la page d'accueil de la dashboard, avec un message flash indiquant l'erreuur.

2.1.1.3 DashBoardController

Le DashboardController gère tout ce qui est commun aux utilisateurs qui ont des profils de joueur. L'ensemble des opérations d'un joueur est accessible à travers ce contrôleur qui dispose de trois attributs d'initialisation dans son contrôleur :

— EntityManagerInterface $entityManagerInterface : L'interface permettant notamment de récupérer les différentes entités utilisées par le contrôleur en passant par leur id et d'autres attributs définis.

— NotificationService : Le service est utilisé pour effectuer toutes les opérations sur les notifications allant de la création à l'affichage des notifications d'un joueur.
— Security $security : L'interface utilisé en symfony pour gérer la sécurité de l'application, l'authentification. Il nous permet aussi de récupérer l'utilisateur actif au moment où il est appelé.

profile() :
Route : /dashboard/profile

Cette route permet d'afficher le profil de l'utilisateur connecté. Toutes les informations concernant l'utilisateur peuvent être retrouvées dans cette route. La fonction vérifie qu'un utilisateur est connecté, puis il récupère toutes les informations nécessaires pour le profil de l'utilisateur : les tables en cours, les tables en attente, les jeux favoris de l'utilisateur et toutes les informations pouvant être récupérés grâce à l'interface security.

Nous utilisons la repository de l'entité Board nommé UserRepository. Dans cette repository, on trouve les fonctions qui nous permettent de manipuler l'entité Board en faisant des requêtes sur cette entité.

La fonction de récupération de Board en fonction du statut de la Board et des utilisateurs participants. C'est avec cette fonction implémentée dans le repository de l'entité Board que nous utilisons pour récupérer les Boards qui sont envoyés dans le template de cette route.

profile(Int) :
Route : /dashboard/profile/userid

Cette route permet de voir le profil des autres utilisateurs présents dans la plateforme.

Nous utilisons les mêmes fonctions implémentés dans la fonction profile() précèdent pour réaliser cette route qui contient les mêmes données et le même template que celui de /dashboard/profile.

boardUser(Request, BoardRepository) :
Route : /dashboard/user

La fonction boardUser permet d'obtenir toutes les Boards dans lesquels l'utilisateur participe. Il y a aussi le système de filtre qui est ajouté a cette route pour permettre de sélectionner parmi les Boards disponibles.

Parmi les paramètres de la fonction boardUser, nous avons la Request qui nous permet d'avoir accès à des composants comme un formulaire. Nous retrouvons aussi le BoardRepository qui nous permet de manipuler les Boards du coté métier.

L'intégration du modèle de donnée de recherche aussi figure comme la base de notre système de filtre qui manipule les données des Boards.
On intègre aussi le type de formulaire créé pour le filtrage des Boards. C'est la classe SearchBoardType placé dans le dossier Form. Cette classe contient les différents entrées de données qui constituent chacune un type de recherche bien identifié dans le STB.

La fonction boardUser récupère d'abord les Boards dans lesquels l'utilisateur connecté participe pour le renvoyer au template. Par la suite elle crée une instance du modèle de donnée de recherche qu'elle injecte à un formulaire par la fonction createForm. En même temps le type de formulaire créé est aussi injecté dans cette même fonction createForm. Et cette dernière permet de récupérer les données fournies au formulaire pour l'ajouter dans l'instance du modèle de donnée de recherche créé. Ainsi, une fois les données reçues, l'instance va être utilisée comme paramètre pour effectuer concrètement la recherche ou le filtre voulu. Le filtre se fait

par la fonction searchBoards implémentée dans le repository de l'entité Board.

allBoards(Request, BoardRepository) :
Route : /dashboard/tables

La fonction allBoards permet d'obtenir toutes les Boards créés dans la plateforme. Il y a aussi le système de filtre qui est ajouté a cette route pour permettre de sélectionner parmi les Boards disponibles.

Les mêmes paramètres qui sont fournis à la fonction BoardUser sont fournis à cette fonction allBoards.

L'intégration du modèle de donnée de recherche aussi figure comme la base de notre système de filtre qui manipule les données des Boards.
On intègre aussi le type de formulaire créé pour le filtrage des Boards. C'est la classe SearchBoardType placé dans le dossier Form. Cette classe contient les différents entrées de données qui constituent chacune un type de recherche bien identifié dans le STB.

On a les mêmes réalisations pour la fonction allBoards que pour boardUser. La seule différence se trouve sur les types de Boards renvoyés. Pour allBoards, on a tous les Boards de l'application qui sont récupérés alors que pour BoardUser c'est seulement ceux sur lesquels l'utilisateur connecté participe.

tablesByGames(Request, BoardRepository, Int) :
Route : /dashboard/game/id/tables

La fonction tablesByGame permet de recupérer toutes les Boards d'une jeu bien donné.

Parmi les paramètres de la fonction tablesByGames, nous avons la Request qui nous permet d'avoir accès à des composants comme un formulaire pour la recherche de Boards. Nous retrouvons le BoardRepository qui nous permet de manipuler les Boards du coté métier. Et le paramètre le plus déterminant dans cette fonction est l'identifiant du jeu pour lequel on cherche ses tables.

L'intégration du système de recherche de Boards aussi c'est fait dans cette route. La seule différence qui est noté est que dans les autres routes c'est la fonction de recherche utilisé. Dans allBoards et boardUser, on utilise la fonction searchBoards alors que dans celle ci on utilise searchboardsByGame qui en plus de ce que fait searchBoards réalise le filtrage qu'en fonction du jeu donné en paramètre.

La fonction tablesByGames récupère d'abord tous les Boards d'un jeu bien donné en paramètre pour le renvoyer au template. Par la suite elle crée une instance du modèle de donnée de recherche qu'elle injecte à un formulaire par la fonction createForm. En même temps le type de formulaire créé est aussi injecté dans cette même fonction createForm. Et cette dernière permet de récupérer les données fournies au formulaire pour l'ajouter dans l'instance du modèle de donnée de recherche créé. Ainsi, une fois les données reçues, l'instance va être utilisé comme paramètre pour effectuer concrètement la recherche ou filtre voulu. Le filtre se fait par la fonction searchBoardsByGame implémenté dans le repository de l'entité Board.

2.1.1.4 GameController

2.1.1.5 HomeController

2.1.1.6 PublisherController

2.1.1.7 RedirectController

Le RedirectController gère la redirection des utilisateurs selon leur rôle. Il est chargé de vérifier le statut de l'utilisateur pour le rediriger vers la partie qui le concerne dans la plateforme. Il est créé pour gérer la position par défaut de l'utilisateur dans la plateforme une fois qu'il se connecte.

index(Security) :
Route : /redirect

Cette fonction permet de récupérer le rôle de l'utilisateur et à partir de cela de savoir dans quelle route il doit l'acheminer

La fonction prend comme paramètre l'interface Security qui nous permet de recupérer l'utilisateur connecté et le manipuler comme voulu.

La Fonction récupère d'abord l'utilisateur connecté grâce à l'interface Security, puis il récupère les rôles de l'utilisateur. Et une fois qu'il a les rôles elle vérifie si le rôle de joueur est parmi ces rôles. Si tel est le cas elle redirige l'utilisateur vers la partie dashboard d'un utilisateur sinon elle le redirige vers la partie administrateur de la plateforme. Cette route est principalement utilisé dans le fichier de configuration de l'authentification qui est Security.yaml.

2.1.1.8 SecurityController

Le SecurityController prend en charge tout ce qui est authentification dans l'application. C'est lui qui gère la connexion d'un utilisateur et sa déconnexion. Quand un utilisateur veut se connecter c'est ce controleur qui entre en jeu.

login(AuthenticationUtils) :
Route : /login

La fonction login permet à tous les utilisateurs du système de se connecter. Elle est appelé à chaque fois que l'utilisateur cherche à se connecter.

La fonction prend en paramètre AuthenticationUtils qui est un composant de sécurité proposé par le framework et qui fournit des méthodes utilitaires pour faciliter le travail avec le processus d'authentification dans nos contrôleurs. Ce paramètre simplifie le système d'authentification.

La Fonction vérifie d'abord si un utilisateur est connecté ou pas pour éviter qu'on ait une double authentification dans l'application pour éviter des problèmes de synchronisation. Si un utilisateur est déjà connecté, elle nous redirige vers l'acceuil de l'application sinon elle laisse passer l'authentification se faire. La fonction récupère le dernier nom d'utilisateur donné et l'injecte dans le formulaire. Elle récupère aussi les erreurs qui peuvent se créées pendant l'authentification et les renvoie dans le template d'authentifcation pour que l'utilisateur sache ce qui a causé le fait qu'il ne s'est pas bien connecté.

logout() :
Route : /logout

La fonction logout permet de se déconnecter. Elle est appelé à chaque fois que l'utilisateur cherche à se déconnecter.

La fonction ne prend pas de paramètre.

Elle lance une exception pour détruire toute objet d'utilisateur contenu dans le système de gestion d'utilisateur du framework.

2.1.1.9 RegistrationController

Ce contrôleur peut être responsable de l'affichage du formulaire d'inscription initial. Cela peut inclure la création d'un formulaire d'inscription à l'aide de la classe RegistrationFormType. Il est également responsable des soumissions de formulaire d'inscription. Cela implique de recevoir les données du formulaire soumis par l'utilisateur, de les valider et, si elles sont valides, de créer un nouveau compte utilisateur dans la base de données. Elle gère aussi la validation du compte par l'émail de vérification. Il n'a qu'un seul paramètre de construction :

— EmailVerifier $emailVerifier : c'est un composant de sécurité de symfony qui permet de vérifier si l'adresse e-mail fournie par l'utilisateur est valide et unique. Cela peut impliquer l'envoi d'un e-mail de confirmation à l'adresse fournie, contenant un lien unique que l'utilisateur doit cliquer pour confirmer son adresse e-mail

register(Request, UserPasswordHasherInterface, EntityManagerInterface) :
Route : /register

Cette fonction gère le processus d'inscription des utilisateurs, y compris la création du formulaire, le traitement de la soumission du formulaire et le rendu du formulaire dans le template Twig.

La fonction prend en paramètre le Request qui permet de gérer le formulaire d'inscription, le UserPasswordHasherInterface qui permet de hacher le mot de passe qui sera fourni par l'utilisateur et le EntityManagerInterface qui permet notamment de récupérer les différentes entités utilisées par le contrôleur.

La Fonction vérifie d'abord si un utilisateur est déjà connecté. Si c'est le cas, l'utilisateur est redirigé vers la page d'accueil. Ensuite, elle crée un formulaire d'inscription en utilisant le type de formulaire RegistrationFormType qui est une classe configuré dans le dossier Form. Elle traite la soumission du formulaire par l'utilisateur. Si le formulaire est soumis et valide, elle extrait les données du formulaire, comme le mot de passe et l'adresse e-mail, et crée un nouvel objet utilisateur. Elle hache le mot de passe fourni par l'utilisateur avant de le stocker dans la base de données. Cela se fait en utilisant l'interface UserPasswordHasherInterface fournie par Symfony. Elle attribue un rôle par défaut à l'utilisateur nouvellement créé. L'utilisateur reçoit le rôle ROLE USER. Elle persiste les données de l'utilisateur dans la base de données en utilisant l'EntityManager. Elle envoie un e-mail de confirmation à l'utilisateur nouvellement inscrit, contenant un lien pour vérifier son adresse e-mail. Cela se fait en utilisant le service EmailVerifier, qui génère et envoie l'e-mail de confirmation. Si le formulaire est soumis mais invalide, elle récupère les erreurs de validation et les affiche à l'utilisateur.

verifyUserEmail(Request, TranslatorInterface) :
Route : /verify/email

La fonction verifyUserEmail de ce contrôleur est responsable de la vérification de l'adresse e-mail d'un utilisateur après qu'il a cliqué sur le lien de confirmation envoyé par e-mail.

La fonction prend en paramètre le Request et le TranslatorInterface qui fournit des fonctionnalités de traduction de texte.
L'application envoie des emails dans un serveur de mail interne. Les réalisations de cette fonction ne sont pas indispensables dans l'application pour le système d'authentification.

La fonction vérifie d'abord si l'utilisateur est pleinement authentifié. Cela signifie que l'utilisateur doit être connecté pour pouvoir vérifier son adresse e-mail. Ensuite, la fonction utilise le service EmailVerifier pour valider le lien de confirmation envoyé par e-mail. Elle vérifie si le lien est valide et si oui, elle marque l'utilisateur comme vérifié dans la base de données. Si une exception se produit lors de la validation du lien, la fonction capture l'exception et affiche un message d'erreur approprié à l'utilisateur. Si la validation du lien réussit, la fonction affiche un message de succès à l'utilisateur et le redirige vers une page appropriée de l'application.

2.1.1.10 SecurityController

2.1.1.11 UserController

Ce contrôleur s'occupe de la redirection de toutes les pages et intéractions autour des acitivités liées au compte utilisateur (donc manipulant principalement des entités User). Ses attributus (initialisés dans son constructeur) sont les suivants :

- UserService $userService : Le service liée au UserController, qui va contenir la majorité du code métier qui sera utilisé dans le controlleur
- Security $security : L'interface utilisé notamment pour récupérer l'entité User de l'utilisateur courant via son id
- EntityManagerInterface $entityManager : L'interface permettant notamment de récupérer les différentes entités utilisées par le contrôleur en passant par leur id et d'autres attributs définis
- UserPasswordhasher $userPasswordhasher : L'interface service utilisé pour le hashage sécurisé des mots de passe des utilisateurs
- TokenStorageInterface $tokenStorage : L'interface service utilisé pour stocker et récupérer de manière sécurisée les jetons d'authentification et de session utilisateur

addContact(Int) :
Route : /user/addContact/contact_id

Ce contrôleur correspond à la route utilisée lorsque un joueur souhaite ajouter un autre joueur en contact, depuis le bouton étoile présent sur la page de profil du joueur ajouté.

La fonction va simplement faire appel à la fonction addContact du UserService afin d'ajouter l'user désigné par l'id passé en paramètre, dans la liste des contact de l'user récupéré via le service Security. Si le code de retour de l'UserService est positif, on recharge simplement la page profil (le bouton étoile sera plein, signifiant que l'ajout s'est déroulé correctement). S'il est négatif, on recharge aussi la page, mais avec cette fois un message flash indiquant l'erreur.

removeContact(Int) :
Route : /user/addContact/contact_id

Ce contrôleur correspond à la route utilisée lorsque un joueur souhaite supprimer un joueur de sa liste des contacts, depuis le bouton étoile présent sur la page de profil de ce joueur.

La fonction va simplement faire appel à la fonction removeContact du UserService afin d'ajouter l'user désigné par l'id passé en paramètre, dans la liste des contact de l'user récupéré via le service Security. Si le code de retour de l'UserService est positif, on recharge simplement la page profil (le bouton étoile sera vide,

signifiant que le retrait s'est déroulé correctement). S'il est négatif, on recharge aussi la page, mais avec cette fois un message flash indiquant une erreur.

contacts(Int) :
Route : /contacts

Ce contrôleur correspond à la route utilisée lorsque l'on souhaite charger le template twig affichant la liste des contacts d'un joueur.

La fonction récupère simplement l'utilisateur courant, ainsi que sa liste des contacts, et va réaliser un rendu du template twig *platform/dashboard/contacts.html.twig*, en envoyant la liste des contacts récupérée.

editAccount(Request, User) :
Route : /dashboard/id/editProfile & /admin/id/editProfile

Cette fonction est responsable de la modification des données du profil utilisateur, à la fois pour les joueurs, ainsi que pour le compte administrateurs. Elle prend 2 routes différentes selon si c'est un joueur ou l'administrateur qui souhaite accéder à cette page

La fonction va d'abord définir sa route de destination selon la route d'arrivée du controlleur (donc selon si l'utilisateur est joueur ou administrateur). Elle crée ensuite un formulaire de type EditProfileType, et l'affiche sur la nouvelle page chargée. Une fois que l'utilisateur a rempli le formulaire, la fonction va récupérer les données du formulaire via le paramètre Request. Elle vérifie d'abord que le formulaire a bien été soumis, puis modifie le mot de passe de l'entité User passé en paramètre, et enfin redirige l'utilisateur sur la route de destination calculée précédement avec un message flash de succès.

En cas d'échec de validation du formulaire, la redirection se fait sur cette même route calculée, mais avec cette fois un message flash d'erreur.

deleteUser(Int, SessionInterface) :
Route : /user/delete/id

Cette fonction gére la suppression d'un utilisateur en fonction de l'identifiant fourni en paramètre.

Elle commence par récupérer l'utilisateur correspondant à l'identifiant fourni en utilisant l'EntityManager. Elle supprime ensuite le compte et réactualise la base de données. Pour garantir que l'utilisateur est déconnecté après sa suppression, la fonction invalide le token de sécurité de l'utilisateur en le réinitialisant à null. Un message de succès est ajouté à la session flash pour informer l'utilisateur que la suppression de son compte a été effectuée avec succès. L'utilisateur est à la fin rediriger vers la page d'accueil du site.

autoDeleteUser(Int, SessionInterface) :
Route : /user/autodelete

Cette fonction permet à un utilisateur de supprimer son propre compte de manière automatique.

La méthode commence par récupérer l'utilisateur actuellement connecté. Si aucun utilisateur n'est trouvé (c'est-à-dire si aucun utilisateur n'est connecté), la fonction redirige l'utilisateur vers la page d'accueil.

Une fois l'utilisateur récupéré, la fonction le supprime de la base de données en utilisant l'EntityManager. Pour garantir que l'utilisateur est déconnecté après sa suppression, la fonction invalide le token de sécurité

de l'utilisateur en le réinitialisant à null. Un message de succès est ajouté à la session flash pour informer l'utilisateur que la suppression de son compte a été effectuée avec succès. L'utilisateur est à la fin rediriger vers la page d'accueil du site.

2.1.2 Entity/Platform

2.1.2.1 Board

2.1.2.1.1 Description

L'entité Board représente le concept de table de jeu sur la plateforme. Les joueurs (User) rejoignent une table (Board), qui est associé à un jeu (Game) dont il va lancer une instance de partie une fois que toutes les places de la table sont occupés par des joueurs. Un Board ne peut lancer qu'une partie d'un Game, et est archivé à la fin de celle-ci.

2.1.2.1.2 Attributs

- **id** : Identifiant unique auto-généré de la table, représentant l'identifiant de la table.
- **nbUserMax** : Le nombre maximum d'utilisateurs autorisés dans la table
- **status** : Le statut actuel de la table, qui peut être "WAITING" (en attente de joueurs avant lancement), "IN_GAME" (en cours de partie) ou "FINISHED" (partie terminé et table archivé)
- **creationDate** : La date de création de la table
- **invitationTimer** : La date butoire avant qu'une invitation à une table n'expire (enregistré au format DateTime).
- **inactivityTimer** : Le délai d'inactivité avant qu'un utilisateur puisse être exclue de la table par les autres joueurs. Le joueur a donc jusqu'à cette date pour jouer son coup. Elle est recalculée après chaque coup, selon l'attribut inactivityHours.
- **nbInvitations** : Le nombre d'invitations qui ont été envoyés par l'utilisateur.
- **listUser** : La collection des joueurs (User) qui sont actuellement dans la table.
- **inactivityHours** : Le nombre d'heures disponible pour chaque joueur pour jouer un coup.
- **game** : Le jeu (Game) associée à la table, représentée par une relation Many-to-One avec l'entité Game.
- **partyId** : L'identifiant de l'instance de la partie du Game à laquelle la table est associée. C'est cette identifiant qui est notamment utilisée lors des communications avec les services et contrôleurs de la partie "Jeux" de la plateforme Agora.
- **invitedContacts** : La collection représentant les utilisateurs invités à rejoindre la table, avec une relation Many-to-Many avec l'entité User.
- **STATUS_WAITING, STATUS_IN_GAME et STATUS_FINISHED** : Ce sont les identifiants qui décrivent chaque statut que peut avoir une table. Elles permettent surtout pour un contrôleur ou service de pouvoir utiliser les statuts sans forcément connaître la valeur du string associée au statut.

2.1.2.1.3 Relation avec les autres entités

- Game : Un Board est associé à un unique Game (et peut seulement lancer une seule partie de ce Game). Evidemment, plusieurs Board peuvent être associés au même Game.
- User : Un Board va posséder 2 types d'associations Many To Many avec des entités User : la première décrivant les joueurs inscrits à la table (qui peuvent rejoindre plusieurs table en même temps), et la deuxième décrivant les joueurs invités à la table (qui peuvent être invité à plusieurs table en même temps).

2.1.2.2 Game

2.1.2.2.1 Description

L'entité Game représente le concept de jeu sur la plateforme. Elle va contenir notamment les données descriptives du jeu (les informations autour du jeu), ainsi que certaines valeurs qui seront utilisés pour communiquer avec les contrôleurs et services de la partie "Jeux" de la plateforme Agora. Un Board va pouvoir s'associer à un Game, duquel il va pouvoir en instancier une seule partie que des Users pouront rejoindre pour jouer.

2.1.2.2.2 Attributs

- **id** : Identifiant unique auto-généré de la partie, représentant l'identifiant du jeu

— **name** : Le nom du jeu
— **descrRule** : Un court texte de description de jeu (présent notamment sur les cartes de prévisualisation du jeu)
— **imgURL** : Le chemin vers l'image de couverture du jeu utilisé par la partie front du site
— **label** : Le libéllé du jeu, utilisé par les controlleurs et services de la partie "Jeux" de la plateforme pour les identifier
— **isActive** : Un booléen permettant de définir si le jeu est disponible ou non
— **minPlayers** : Le nombre minimum de joueurs
— **maxPlayers** : le nombre maximum de joueurs
— **inactivityHours** : Le nombre d'heures disponible pour chaque joueur pour jouer un coup.
— **boards** : La collections des Board s'étant associé au Game.

2.1.2.2.3 Relation avec les autres entités

— Board : Une relation One To Many permet de mettre en relation un Game et les Boards qui s'y sont associés. Cette relation est notamment utilisée pour assurer le filtrage des tables par jeu.

2.1.2.3 User

2.1.2.3.1 Description

L'entité User représente le concept de compte joueurs sur la plateforme. Un utilisateur qui souhaite joueur sur la plateforme va se créer un compte joueur, et ses données seront donc enregistrées sous une entité User. L'entité va donc contenir l'ensemble des données du compte joueur.

2.1.2.3.2 Attributs

— **id** : Identifiant unique auto-généré de la partie, représentant l'identifiant de l'utilisateur
— **email** : L'email renseigné par l'utilisateur lors de son inscription
— **roles** : Les différents rôles de l'utilisateur dans le système (ROLE_ADMIN pour l'administrateur, ROLE_USER pour un joueur et ROLE_MODERATOR pour un modérateur).
— **password** : Le mot de passe de l'utilisateur, stocké sous forme hashée
— **username** : Le nom d'utilisateur de l'utilisateur, avec une contrainte de longueur entre 4 et 20 caractères.
— **isVerified** : Un booléen indiquant si l'utilisateur a bien fait vérifier son adresse mail via l'email de vérification envoyé à la création du compte.
— **favoriteGames** : Une collection de Game représentant les jeux préférés de l'utilisateur
— **boards** : La collection des tables auxquelles l'utilisateur est inscrit
— **notifications** : Une collection des entités Notification reçues par l'utilisateur
— **contacts** : Une collection représentant les contacts enregistrés par l'utilisateur
— **isBanned** : Un booléen indiquant si l'utilisateur est banni ou non
— **createdAt** : La date de création du compte joueur

2.1.2.3.3 Relation avec les autres entités

— Board : Une relation Many To Many représentant les tables où s'est inscrit l'utilisateur (un utilisateur peut s'inscrire à plusieurs tables, et plusieurs utilisateurs peuvent s'inscrire à la même table).
— Game : Une relation Many To Many représentant les jeux favoris de l'utilisateur (un utilisateur peut mettre en favori plusieurs jeux, et plusieurs utilisateurs peuvent avoir en favori le même jeu).
— Notification : Une relation One To Many représentant les notifications reçus par l'utilisateur (un utilisateur peut recevoir plusieurs notifications, mais une notification ne s'envoie qu'à un seul utilisateur).

2.1.3 Form/Platform

2.1.3.1 BoardRegistrationType

Cette classe est utilisée pour générer le formulaire d'inscription d'un utilisateur à une table. Ses attributs sont :

— EntityManagerInterface $entityManagerInterface : L'interface permettant notamment de récupérer les différentes entités utilisées par le contrôleur en passant par leur id et d'autres attributs définis.
— Security $security : L'interface utilisé en symfony pour gérer la sécurité de l'application, l'authentification. Il nous permet aussi de récupérer l'utilisateur actif au moment où il est appelé.

buildForm(FormBuilderInterface, Array) :

C'est cette fonction qui va s'occuper de l'instanciation du formulaire. Le paramètre Array va nous permettre ici de passer l'entité Game qui sera associée à la futur Board.

Elle récupère d'abord la liste de tous les utilisateurs (en excluant l'administrateur), puis elle configure un tableau qui va contenir l'ensemble des choix possibles pour le nombre de joueurs (le minimum et le maximum pour ce nombre est récupéré depuis l'attribut minPlayer et maxPlayer du Game passé en paramètre). Elle utilise ensuite le FormBuilder passé en paramètre afin de créer le formulaire, avec comme champ pour celui ci

— **nbUserMax** : Un ChoiceType pour le nombres de joueurs de la table (utilisant le tableau crée précédemment pour définir les choix possibles)
— **invitedContacts** : Un EntityType d'où l'utilisatuer pourra choisir les joueurs qu'il souhaite inviter à sa table
— **inactivityHours** : Un NumberType d'où l'on pourra renseigner le nombre d'heures disponibles pour chaque joueur pour joueur un coup (de 24 à 168 heures).

configureOption(OptionsResolver) :

C'est dans cette fonction que l'on indique les paramètres attendus par la fonction buildForm afin de créer le formulaire. Pour ça, les paramètres sont indiquer dans le tableau passé en paramètre lors de l'appel à la méthode setDefault de l'OptionResolver (data_class représentant le type d'entité crée par le formulaire, et 'game' le paramètre que l'on utilise pour passer l'entité Game qui sera associé à la table).

2.1.4 Service/Platform

2.1.4.1 BoardManagerService

Ce service va contenir la majorité du code métier agissant sur les entités Board de la plateformee. Ses attributs sont :

- GameManagerService $gameManagerService : Le service permettant d'agir sur la partie qui sera associé au Board (décrit dans le document *"Document technique jeux : Description de la bibliothèque de code*)
- EntityManagerInterface $entityManagerInterface : L'interface permettant notamment de récupérer les différentes entités utilisées par le controlleur via leur id
- NotificationService $notificationService : Le service permettant l'envoi de notifications aux joueurs sur la plateforme.
- $SUCCESS : Attribut statique correspondant au code de retour d'une fonction du service en cas de succès.
- $DAYS_BEFORE_EXPIRATION : Attribut statique de type string utilisé pour calculer la date d'expiration d'une invitation (1 semaine après son envoi)

setUpBoard(Board, Game) :

Cette fonction permet d'initialiser un Board, avant sa persistance dans la base de données et après le renseignement des paramètres de la partie par le joueur créant le Board, et ce depuis un formulaire de type BoardRegistrationType.

La fonction va d'abord initialiser la date de création (CreationDate) du Board (en utilisnt la date et l'heure de l'appel à la fonction) et la date d'expiration des invitations qui ont pu être fait par le joueur.

Elle crée ensuite la partie du Game que le joueur veut joueur (via l'utilisation du GameManagerService, en appelant sa fonction createGame et en passant en paramètre le label du Game), et renseigne l'id du Game dans le Board, ainsi que l'id de la partie instanciée (retournée par createGame) , afin de pouvoir les récupérer ultérieurement. Elle retourne enfin le code de succès $SUCCESS

addUserToBoard(Board, User) :

Cette fonction permet de pouvoir ajouter un User à un Board. Le Board passé en paramètre ne doit pas être plein (Board->isFull())

La fonction ajoute d'abord l'User à la partie, via la fonction joinGame du GameManagerService (en passant en paramètre l'id de la partie, ainsi que l'objet entité du User), puis ajoute l'User à la liste des joueurs du Board. Elle fait aussi la vérification de si l'utilisateur était invité à la table afin de le retirer de la liste des joueurs invités, pour gérer le cas où l'utilisateur a été invité à cette table, mais qu'il n'est pas passé par le lien d'invitation

Ensuite, elle vérifie si le joueur qui a rejoint a complété la table, auquel cas on appel la fonction launchGame du GameManagerService (en passant en paramètre l'id de la partie) afin de lancer la partie. On applique l'état "IN_GAME" au Board, puis on notifie tout les joueurs du démarrage de la partie via le NotificationService.

Enfin, peut importe le cas, elle procède à une persistance des nouvelles données dans la base de données, puis ajoute le Board à la liste des boards du User. Elle retourne enfin le code de succès $SUCCESS

removePlayerFromBoard(Board, User) :

Cette fonction permet de pouvoir retirer un User d'un Board, ainsi que de la partie liée au Board.

La fonction retire d'abord le User de la liste des joueurs du Board, puis appel le GameManagerService afin de retirer l'User de la partie. Elle effectue ensuite une persistance des nouvelles données dans la base de données, puis renvoie le code de succès $SUCESS.

2.1.4.2 GameViewerService

Ce service est l'un des plus importants dans le projet Agora, puisqu'il est celui permettant d'afficher la vue des parties depuis l'interface web. Il est donc celui qui fait la passerelle entre le code Plateforme et le code jeux d'Agora. Son seul attribut est $routesArray, qui est un tableau associatif [String ; String]. Celui-ci permet, à partir du label d'un Game utilisé comme clé du tableau, de retrouver la route d'affichage d'une instance de partie de ce Game.
Une clé de $routesArray doit donc être composé de seulement 3 caractères (lettres ou chiffres) en majuscule, à savoir le label d'un Game, et la valeur associée doit être le nom de la route d'affichage de la vue de ce Game (tel que défini dans le document *"Document technique jeux : Description de la bibliothèque de code*). C'est dans le constructeur que l'on réalise l'ajout d'un nouveau jeu dans la table.

getGameViewRouteFromLabel(String) :

Cette fonction retourne la route du Game dont le label est passé en paramètre de cette fonction. La route récupérée peut ensuite être utilisée pour afficher une partie du Game, en utilisant l'id de la partie généré par le GmaeViewerService lors de l'appel à la fonction CreateGame.

2.1.4.3 NotificationService

Ce service gère les notifications dans l'application. Elle fournit des méthodes pour envoyer des notifications à des utilisateurs individuels ou à des groupes d'utilisateurs, stocker des notifications dans la base de données et récupérer les notifications non lues d'un utilisateur. Son constructeur initialise deux paramètres qui sont indispensables pour l'envoi et la réception des notifications en temps réel. Ce sont

- $hub (HubInterface) : Interface du hub Mercure utilisé pour publier les mises à jour.
- $entityManager (EntityManagerInterface) : Interface de gestion des entités Doctrine utilisée pour persister les notifications dans la base de données.

notifyGroup(List, String, Date) :

Cette fonction envoie une notification à un groupe de clients abonnés à un sujet spécifique via Mercure qui est un protocole et un système de messagerie en temps réel qui permet aux serveurs de publier des mises à jour (updates) vers des clients HTTP/2, WebSocket ou Server-Sent Events (SSE), de manière asynchrone et bidirectionnelle. Les paramètres de cette fonction sont :

- $users : Un tableau d'objets utilisateur à notifier
- $content : Le contenu de la notification
- $date : La date de la notification

On utilise le $hub du HubInterface et la fonction update de mercure pour créer le topic qui sera publié avec le contenu passé en paramètre.

notifyUser(String, String, Date) :

La fonction notifyUser envoie une notification à un utilisateur spécifique. Les paramètres de cette fonction sont :

- $users : Un tableau d'objets utilisateur à notifier
- $content : Le contenu de la notification
- $date : La date de la notification

On utilise aussi le $hub du HubInterface et la fonction update de mercure pour créer le topic qui sera publié avec le contenu passé en paramètre. Elle est utilisé à chaque fois qu'un nouveau message est créé.

storeNotification(User, String) :

La fonction stocke une notification dans la base de données. Les paramètres qui la définissent sont :

- $user : L'utilisateur destinataire de la notification
- $content : Le contenu de la notification

On crée l'instance de la notification et on donne à ses attributs des valeurs. Et par la suite, on utilise l'interface de gestion des entités EntityManager pour persister la notification créée dans la base de donnée. Elle est utilisée à chaque fois qu'une notification se déclenche dans la plateforme.

notifyManyUser(List, String, Date) :

Cette fonction envoie une notification à plusieurs utilisateurs et stocke les notifications dans la base de données. Les paramètres sont celles des fonctions précédentes :

- $users : Un tableau d'objets utilisateur à notifier
- $content : Le contenu de la notification
- $date : La date de la notification

Elle utilise les deux fonctions décrites précedemment : notifyUser et storeNotification pour notifier chaque utilisateur concerné. Elle est utilisé au lancement de chaque table de jeu qui commence.

getNotifications(Security) :

Cette fonction récupère les notifications non lues d'un utilisateur à partir de la base de données. Elle prend en paramètre :

- $security : Le service de sécurité fourni par le framework

Elle récupère l'utilisateur connecté. S'il y a un utilisateur actif, elle manipule l'interface de gestion des entités EntityManager pour recupérer l'ensemble des notifications non lues de l'utilisateur courant.

2.1.4.4 UserService

Cette classe service va regrouper les fonctions métiers utilisés notamment par les contrôleurs regroupés dans la classe UserController. Ses attributs sont :

- $entityManager (EntityManagerInterface) : Interface de gestion des entités Doctrine utilisée pour persister les notifications dans la base de données.
- $ADD_HIMSELF_ERROR : Code statique définissant le code d'erreur lorsque un utilisateur s'ajoute lui même comme contact
- $ALREADY_CONTACT_ERROR : Code statique définissant le code d'erreur lorsque un utilisateur ajoute un autre User qu'il a déjà en contact
- $NOT_A_CONTACT_ERROR : Code statique définissant le code d'erreur lorsque un utilisateur essaye de retirer un User de sa liste des contacts alors qu'il n'était pas présent
- $SUCCESS : Code statique définissant le code de réussite d'une fonction de ce service

addContact(Int, Int) :

Cette fonction permet de pouvoir ajouter un User à la liste des contacts de l'User courant ; Le premier paramètre correspond à l'id de l'User courant, tandis que le deuxième correspond à celui de l'User qu'il souhaite ajouter.

La fonction vérifie d'abord que les deux id ne sont pas les mêmes (auquel cas elle renvoie le code d'erreur associé), et que le futur contact n'est pas déjà présent dans la liste (en récupérant les entités concernés via l'EntityManager) (auquel cas elle renvoie le code d'erreur associé). La fonction procède ensuite à l'ajout du nouveau contact dans la liste, persiste les nouvelles données de l'User courant, et renvoie le code de succès

removeContact(Int, Int) :

Cette fonction permet de pouvoir retirer un User à la liste des contacts de l'User courant ; Le premier paramètre correspond à l'id de l'User courant, tandis que le deuxième correspond à celui de l'User qu'il souhaite ajouter.

La fonction vérifie d'abord que l'User dont l'on souhaite le retrait de la liste est bien présent dans cette dernière (sinon elle renvoie le code d'erreur associé). Elle procède ensuite au retrait du contact de la liste, persiste les nouvelles données de l'User courant, et renvoie le code de succès

2.1.5 TwigExtension/Platform

2.1.5.1 BoardFilterExtension

Cette classe sert à créer un filtre twig personnalisé, utilisé pour les notifications d'invitations à une table. La fonction getFilters permet d'associer le nom du filtre tel qu'il va être appelé avec twig, avec la fonction de filtre (en passant le nom de la fonction). La fonction getName est obligatoire, mais ne fait que renvoyer le nom du filtre sur twig.

Enfin , la fonction formatBoardId décrit le comportement de notre filtre. Elle va extraire l'identifiant de la table à partir du contenu d'une notification d'invitation à une table. Elle prend en paramètre le message de la notification et utilise une expression régulière pour rechercher un motif correspondant à un identifiant de table entre parenthèses. Si un identifiant est trouvé, il est retourné en tant qu'entier. Sinon, la méthode retourne 0.

Ce filtre nous est nécessaire pour récupérer l'id du board présent dans le contenu textuel de la notification, afin de la passer en paramètre de la route du contrôleur checkInvitations (présent dans le fichier BoardController).

2.2 templates/platform

2.2.1 admin

2.2.1.1 Dashboard administrateur (layout.html.twig)

Cette page est très similaire à la page Dashboard/layout.htmltwig (voir 2.2.2.1), mais sans le système de notification. Elle offre une navigation vers différentes sections telles que :

— **Accueil** : Cette page affiche le gestionnaire des utilisateurs (voir 2.2.1.5).
— **Générateur de compte** : Cette page affiche le générateur de compte (voir 2.2.1.6).
— **Gestion des bannissements** : Cette page affiche le gestionnaire des bannissements des utilisateur. (voir 2.2.1.2).
— **Gestion des tables** : Cette page affiche le gestionnaire des tables de jeu. (voir 2.2.1.4).

De plus, elle inclut un lien vers la page paramètre de l'administrateur (voir 2.2.8).

2.2.1.2 Gestion des bannissements (banmanager.html.twig)

Cette page est un tableau de gestion des bannissements, qui permet à un administrateur de visualiser et de gérer les utilisateurs bannis ou non sur la plateforme.

2.2.1.2.1 Contenu

1. **Sections de recherche**
   — **Recherche par utilisateur** : Un champ de texte est fourni pour que l'utilisateur puisse saisir le nom d'un utilisateur à rechercher.
   — **Recherche par email** : Un autre champ de texte est fourni pour rechercher un utilisateur par son adresse email.
2. **Tableau des utilisateurs bannis**
   — **Structure du tableau** : Chaque rangée du tableau représente un utilisateur et contient les informations pertinentes sur cet utilisateur.
   — **Colonnes du tableau** :
      — **Profil** : Cette colonne affiche l'avatar de l'utilisateur.
      — **Identifiant** : Affiche le nom d'utilisateur de l'utilisateur.
      — **Email** : Affiche l'adresse email de l'utilisateur.
      — **État** : Indique si l'utilisateur est actuellement banni ou non.
      — **Action** : Cette colonne contient des liens d'action pour chaque utilisateur, tels que "Bannir", "Débannir" ou "Prolonger" le bannissement.

2.2.1.2.2 JavaScript

Un script JavaScript est inclus à la fin du contenu pour ajouter une fonctionnalité de recherche dynamique. Ce script permet à l'utilisateur de rechercher des utilisateurs bannis en temps réel en fonction des critères de recherche saisis dans les champs de texte.

2.2.1.3 bannissement d'un utilisateur (banUser.html.twig)

Cette page est un formulaire bannissement d'un utilsateur, qui permet à un administrateur de bannir un utilisateur, selon un motif et un délais de bannissement.

1. **Formulaire de bannissement** :
   — Un formulaire est inclus pour permettre à l'administrateur de saisir les détails du bannissement.
2. **Champs de formulaire** :
   — **Motif** : Un champ de texte est fourni pour que l'administrateur saisisse la raison du bannissement.
   — **Durée** : Un champ de nombre est inclus pour spécifier la durée du bannissement, accompagné d'un menu déroulant pour sélectionner l'unité de temps (jours, semaines, mois, années).
3. **Boutons d'action** :
   — Deux boutons d'action sont inclus en bas du formulaire pour permettre à l'administrateur de soumettre le formulaire ou de retourner à la page précédente.

4. **Messages flash** :
   — Une inclusion de message flash est effectuée, ce qui permet que des messages d'alerte ou de confirmation peuvent être affichés à cet emplacement après une action de l'utilisateur.

2.2.1.4 Administration des tables (tableadmin.html.twig)

2.2.1.4.1 Menu d'administration

— **Menu :** Le code HTML représente un menu interactif qui s'affiche lorsqu'un utilisateur interagit avec une rangée de tableau spécifique. Le menu est encapsulé dans une balise <tr> avec un identifiant unique dérivé de l'ID du tableau. Il comprend plusieurs sous-menus :
   — **Menu pour exclure un joueur :** Ce sous-menu affiche une liste déroulante des joueurs disponibles et un bouton pour exclure un joueur sélectionné.
   — **Menu pour supprimer la table :** Ce sous-menu contient un bouton pour supprimer complètement la table.
   — **Bouton pour fermer le menu :** Ce sous-menu affiche un simple bouton pour fermer le menu.
   — **Chatlog :** Ce sous-menu affiche un journal de chat avec les messages associés à la table sélectionnée.
   — **Profils des joueurs :** Ce sous-menu affiche les profils des joueurs participants à la table sous forme de tableau.

   Chaque sous-menu est conçu pour faciliter une interaction spécifique avec la table de jeu, offrant ainsi une expérience utilisateur complète et intuitive.

2.2.1.4.2 Interaction JavaScript

L'interaction JavaScript est cruciale pour améliorer l'expérience utilisateur et rendre les fonctionnalités du site web dynamiques et réactives. Le code JavaScript ci-dessous gère diverses interactions utilisateur, telles que la soumission de formulaires, le basculement des menus, l'exclusion de joueurs et bien d'autres encore.

— **Soumission de formulaire de recherche :** La fonction `submitSearchForm()` est appelée lorsqu'un utilisateur soumet le formulaire de recherche. Cette fonction récupère le formulaire par son ID et le soumet, ce qui déclenche une action de recherche sur le serveur.
— **Basculement du menu :** La fonction `toggleMenu(boardId)` est utilisée pour basculer l'affichage d'un menu spécifique identifié par son ID de tableau. Lorsque le menu est activé, la classe `hidden` est retirée, ce qui le rend visible à l'utilisateur. De plus, la classe `no-scroll` est ajoutée au corps de la page pour empêcher le défilement lorsque le menu est ouvert.
— **submitSearchForm() :**
   — Cette fonction est appelée lorsque l'utilisateur soumet le formulaire de recherche. Elle récupère l'élément <form> avec l'ID 'searchForm' et le soumet, déclenchant ainsi une action de recherche sur le serveur.
— **toggleMenu(boardId) :**
   — Cette fonction est utilisée pour afficher ou masquer le menu associé à un tableau spécifique identifié par son ID. Lorsqu'elle est appelée, elle récupère l'élément du menu correspondant à l'ID du tableau et bascule sa visibilité en ajoutant ou en supprimant la classe 'hidden'. De plus, elle gère le défilement de la page en ajoutant ou en supprimant la classe 'no-scroll' sur le corps de la page pour empêcher le défilement lorsque le menu est ouvert.
— **toggleScroll(menu) :**
   — Cette fonction est utilisée pour activer ou désactiver le défilement sur le menu spécifié en basculant la classe 'no-scroll'.
— **getTableId(event) :**
   — Cette fonction est utilisée pour obtenir l'ID du tableau associé à un événement de clic sur une rangée du tableau. Elle prend en paramètre l'événement de clic et traverse les éléments parents jusqu'à trouver l'élément <tr> (ligne du tableau). Ensuite, elle extrait l'ID du tableau à partir de l'attribut 'id' de cet élément.
— **closeMenu(event, boardId) :**
   — Cette fonction est appelée lorsque l'utilisateur clique sur le bouton de fermeture du menu. Elle empêche la propagation de l'événement, puis bascule le menu associé au tableau spécifié par son ID pour le cacher.

— **handleExcludeButtonClick(event) :**
  — Cette fonction est appelée lorsqu'un utilisateur clique sur le bouton d'exclusion d'un joueur dans le menu. Elle récupère le nom de l'utilisateur sélectionné dans la liste déroulante, puis appelle la fonction 'excludePlayer(userName)' pour exclure ce joueur.
— **excludePlayer(userName) :**
  — Cette fonction envoie une requête POST au serveur avec les informations du joueur à exclure. Elle envoie les données au point de terminaison '/excludePlayer' au format JSON. Elle gère également les réponses du serveur, affichant un message de succès ou d'erreur dans la console du navigateur.
— **Gestion du défilement du menu :** La fonction `toggleScroll(menu)` bascule la classe `no-scroll` pour activer ou désactiver le défilement sur le menu.
— **Obtention de l'ID du tableau :** La fonction `getTableId(event)` est utilisée pour récupérer l'ID du tableau à partir de l'événement de clic de l'utilisateur sur une rangée du tableau.
— **Fermeture du menu :** La fonction `closeMenu(event, boardId)` est appelée lorsque l'utilisateur clique sur le bouton de fermeture d'un menu. Elle empêche la propagation de l'événement, puis bascule le menu pour le cacher.
— **Exclusion de joueur :** La fonction `excludePlayer(userName)` est utilisée pour exclure un joueur sélectionné du jeu. Elle envoie une requête POST au serveur avec les informations du joueur à exclure.
— **Écouteurs d'événements :** Divers écouteurs d'événements sont configurés pour détecter les actions de l'utilisateur, telles que le clic sur des boutons d'exclusion de joueur, le changement des options de sélection dans les menus déroulants, etc.

2.2.1.5 Gestion des utilisateurs (allusers.html.twig)

La gestion des utilisateurs est une composante essentielle de nombreux systèmes d'information, permettant aux administrateurs de contrôler les autorisations, les rôles et les privilèges des utilisateurs. Dans le contexte spécifique d'une application web, la gestion des utilisateurs est souvent réalisée à travers une interface utilisateur dédiée, permettant aux administrateurs d'afficher, de modifier et de supprimer des comptes d'utilisateur.

La vue Twig `allusers.html.twig` présente une implémentation de cette fonctionnalité en fournissant une interface pour gérer les utilisateurs, y compris leurs profils, identifiants, adresses e-mail, rôles et actions disponibles. Cette vue est conçue pour être intégrée dans une structure de page plus large, typiquement utilisée dans un panneau d'administration ou un tableau de bord.

La structure et le fonctionnement de la vue Twig sont décrits comme suit :

— **Extension de modèle :** La vue étend un modèle de mise en page parent (`layout.html.twig`), ce qui permet d'incorporer cette vue dans une mise en page existante et cohérente avec le reste de l'application.
— **Contenu principal :** Le contenu principal de la page est encapsulé dans le bloc `content`. Il comprend l'en-tête, le tableau des utilisateurs et la pagination.
— **Tableau des utilisateurs :** Le tableau des utilisateurs est présenté avec des colonnes pour le profil, l'identifiant, l'e-mail, le rôle et les actions disponibles. Chaque utilisateur est représenté par une ligne dans le tableau.
— **Boucle Twig :** Une boucle Twig est utilisée pour itérer sur chaque utilisateur dans la liste des utilisateurs. Cela permet d'afficher dynamiquement les informations de chaque utilisateur dans le tableau.
— **Colonnes de données :** Les informations de chaque utilisateur sont affichées dans des cellules de tableau, extraites de l'objet utilisateur. Ces informations incluent le profil, l'identifiant, l'adresse e-mail et le rôle.
— **Menu déroulant des rôles :** Chaque utilisateur est associé à un menu déroulant permettant de sélectionner et de modifier son rôle. Les rôles sont présentés de manière conviviale et peuvent être modifiés en temps réel.
— **Boutons d'action :** Des boutons d'action sont fournis pour chaque utilisateur, offrant des fonctionnalités telles que la sauvegarde des modifications de rôle et la suppression de l'utilisateur.
— **Script JavaScript :** Un script JavaScript est inclus pour gérer la fonctionnalité de sauvegarde du rôle via AJAX. Cela permet une interaction dynamique avec le serveur sans rechargement de page.

— **Inclusion de jQuery :** La première ligne du script inclut la bibliothèque jQuery à partir d'une source externe.
— **Fonction `saveRole` :** Cette fonction prend deux paramètres en entrée : l'ID de l'utilisateur (`userId`) et le nom d'utilisateur (`username`). Elle est responsable de récupérer le nouveau rôle sélectionné par l'utilisateur à partir du menu déroulant correspondant à cet utilisateur.
— **Récupération du rôle sélectionné :** Le rôle sélectionné par l'utilisateur est extrait en utilisant jQuery pour obtenir la valeur du menu déroulant identifié par `roleSelect_(userId)`.
— **Envoi de la requête AJAX :** Une requête AJAX de type POST est envoyée à l'URL spécifiée (`/save-role`). Les données envoyées dans la requête comprennent l'ID de l'utilisateur, son nom d'utilisateur et le nouveau rôle sélectionné.
— **Gestion des réponses :** La fonction `success` est définie pour gérer la réponse du serveur en cas de succès de la requête. De même, la fonction `error` est définie pour gérer les erreurs éventuelles survenues lors de l'envoi de la requête.

Pour personnaliser ou étendre cette vue, des modifications peuvent être apportées à la structure HTML du tableau, aux styles CSS ou aux fonctionnalités JavaScript. Ces modifications peuvent être effectuées en fonction des besoins spécifiques de l'application et des préférences de conception de l'interface utilisateur.

La vue Twig `allusers.html.twig` utilise ce tableau des utilisateurs pour afficher les informations pertinentes à propos de chaque utilisateur. Voici comment les données sont intégrées dans le tableau :

— **Colonnes de données :** Les informations de chaque utilisateur sont extraites de l'objet utilisateur et affichées dans les cellules de tableau correspondantes. Par exemple, le nom d'utilisateur est affiché dans la colonne 'Profil', l'adresse e-mail dans la colonne 'Email', etc.
— **Menu déroulant des rôles :** Chaque utilisateur est associé à un menu déroulant permettant de sélectionner et de modifier son rôle. Les options disponibles sont déterminées dynamiquement à partir du rôle actuel de l'utilisateur.
— **Boutons d'action :** Des boutons d'action sont fournis pour chaque utilisateur, permettant de sauvegarder les modifications de rôle ou de supprimer l'utilisateur. Ces boutons sont liés à des fonctions JavaScript qui interagissent avec le serveur via AJAX pour effectuer les actions correspondantes.

2.2.1.5.1 Description des colonnes du tableau des utilisateurs

Dans cette section, nous allons explorer en détail chaque colonne du tableau des utilisateurs, en expliquant le code Twig utilisé pour les afficher et en fournissant des exemples de code pour les boutons d'action.

Colonne	Description
Profil	Affiche le profil de l'utilisateur, y compris une image d'avatar et son nom d'utilisateur. L'image est cliquable et redirige vers le profil complet de l'utilisateur.
Identifiant	Affiche l'identifiant de l'utilisateur. Il s'agit généralement du nom d'utilisateur.
Email	Affiche l'adresse e-mail associée à l'utilisateur.
Rôle	Affiche le rôle actuel de l'utilisateur sous forme de menu déroulant. Permet de modifier le rôle de l'utilisateur. Les options disponibles sont 'Utilisateur', 'Modérateur' et 'Administrateur'.
Action	Fournit des boutons d'action pour chaque utilisateur, permettant de sauvegarder les modifications de rôle ou de supprimer l'utilisateur.

Table 1 – Description des colonnes du tableau des utilisateurs

— **Profil :** Pour afficher le profil de l'utilisateur, nous utilisons une balise `<a>` avec un lien généré à l'aide de la fonction `path` de Symfony. Voici un exemple de code pour afficher le profil de l'utilisateur :

```
<a href="{{ path('app_other_user_profile', {'user_id': user.id}) }}" class="flex items-center just
    <div style="text-align: center;">
```

```
        <img src="https://avatarfiles.alphacoders.com/244/thumb-244471.png" alt="Avatar de {{ user
        <div class="text-black font-bold">{{ user.username }}</div>
    </div>
</a>
```

— **Identifiant :** Pour afficher l'identifiant de l'utilisateur, nous utilisons simplement une balise `<input>` avec la valeur de l'identifiant de l'utilisateur. Voici un exemple de code pour afficher l'identifiant de l'utilisateur :

```
<input type="text" value="{{ user.username }}" class="bg-transparent border-b-2 border-gray-300 py
```

— **Email :** Pour afficher l'adresse e-mail de l'utilisateur, nous utilisons également une balise `<input>` avec la valeur de l'e-mail de l'utilisateur. Voici un exemple de code pour afficher l'adresse e-mail de l'utilisateur :

```
<input type="text" value="{{ user.email }}" class="bg-transparent border-b-2 border-gray-300 py-2"
```

— **Rôle :** Pour afficher le rôle de l'utilisateur sous forme de menu déroulant, nous utilisons une balise `<select>` avec des options dynamiquement générées en fonction du rôle actuel de l'utilisateur. Voici un exemple de code pour afficher le menu déroulant des rôles :

```
<select class="border-b-2 border-gray-300 py-2" id="roleSelect_{{ user.id }}">
    {% set currentRole = user.roles[0]|replace({'ROLE_USER': 'Utilisateur', 'ROLE_MODERATOR': 'Mod
    <option value="{{ user.roles[0] }}">{{ currentRole }}</option>
    {% for role in ['Utilisateur', 'Modérateur', 'Administrateur'] %}
        {% if role != currentRole %}
            <option value="{{ 'ROLE_' ~ role }}">{{ role }}</option>
        {% endif %}
    {% endfor %}
</select}
```

— **Action :** Pour fournir des boutons d'action permettant de sauvegarder les modifications de rôle ou de supprimer l'utilisateur, nous utilisons des balises `<a>` avec des liens vers les routes Symfony correspondantes. Voici un exemple de code pour afficher les boutons d'action :

```
<a href="#" class="text-white bg-primary hover:bg-blue-400 focus:ring-4 focus:outline-none focus:r
    Sauvegarder
</a>
<a href="{{ path('app_user_delete', {'id': user.id}) }}" class="text-white bg-red-500 hover:bg-red
    Supprimer
</a}
```

2.2.1.6 (generateAccount.html.twig)

2.2.2 dashboard

2.2.2.1 dashboard utilisateur(layout.html.twig)

2.2.2.2 Les notifications (layout.html.twig)

Les notifications jouent un rôle essentiel dans l'expérience utilisateur d'une application. Elles permettent d'informer les utilisateurs des événements importants ou des actions qui nécessitent leur attention. Dans cette section, nous expliquerons comment les notifications sont affichées et gérées dans l'interface utilisateur, ainsi que le processus de suppression via AJAX.

2.2.2.2.1 Structure du menu de notification

— **Menu de notification :** Le menu de notification est conçu pour afficher les notifications non lues de l'utilisateur. Il est accessible à partir d'un bouton dédié situé dans le coin supérieur droit de l'interface utilisateur.

  — **Apparence et comportement :** Le menu de notification est initialement caché et apparaît lorsque l'utilisateur clique sur le bouton de notification. Il affiche une liste de notifications non lues avec des détails tels que le type de notification, le contenu et la date de création.

- **Interaction utilisateur :** Chaque notification est accompagnée d'un bouton "Marquer comme lu" qui permet à l'utilisateur de signaler qu'il a pris connaissance de la notification.
- **Fonctionnalités supplémentaires :** Le menu de notification peut également contenir un lien "Tout marquer comme lu" qui permet à l'utilisateur de marquer toutes les notifications comme lues en une seule fois.

— **Conteneur du menu :** La balise `<div>` avec la classe `hidden origin-top-right` est utilisée pour encapsuler le menu de notification. Cette classe spécifique indique que le menu est initialement caché et positionné dans le coin supérieur droit de la fenêtre. Les autres classes telles que `absolute`, `rounded-md`, `shadow-lg`, etc., sont utilisées pour définir le style du menu, comme son positionnement, son ombre et son espacement.

— **Titre du menu :** La balise `<h2>` avec la classe `text-lg font-semibold` affiche le titre "Mes notifications". Cette classe spécifique définit la taille de police et le poids de la police pour le titre.

— **Contenu des notifications :** Les notifications sont affichées à l'intérieur de la balise `<div>` avec l'identifiant `notif-content`. Cette balise contient une boucle Twig qui parcourt toutes les notifications et les affiche dynamiquement. Chaque notification est représentée par un élément `<div>` avec la classe `notification-item`. Les notifications non lues sont filtrées avec la condition if not. Chaque notification contient des informations telles que le type de notification, le contenu et l'heure de création. Ces informations sont affichées en utilisant des balises HTML appropriées.

— **Bouton "Tout marquer comme lu" :** Le bouton "Tout marquer comme lu" est représenté par un lien `<a>` avec l'identifiant `mark-all-as-read`. Ce lien permet de marquer toutes les notifications comme lues en un seul clic.

2.2.2.2.2 Processus de suppression des notifications via AJAX

— **Suppression des notifications :** Lorsqu'un utilisateur clique sur le bouton "Marquer comme lu" pour une notification spécifique, une requête AJAX est envoyée au serveur pour marquer cette notification comme lue sans recharger la page.
- **Fonction markAsRead :** Le processus de suppression est géré par une fonction JavaScript appelée markAsRead. Cette fonction intercepte l'événement de clic sur le bouton "Marquer comme lu", envoie une requête AJAX au serveur et met à jour l'interface utilisateur en supprimant la notification marquée comme lue du menu de notification.
- **Mise à jour asynchrone :** L'utilisation d'AJAX permet de mettre à jour la liste des notifications sans recharger la page, offrant ainsi une expérience utilisateur fluide et réactive.
- **Gestion des erreurs :** Le script JavaScript gère également les erreurs potentielles lors de l'envoi de la requête AJAX, offrant ainsi une expérience robuste même en cas de problèmes de connexion ou de serveur.

2.2.2.3 Les tables et filtres (tables.html.twig | user.html.twig | profil.html.twig | history.html.twig)

Nous verrons dans cette section l'affichage de la liste des tables sur AGORA. En effet, ces tables sont disponibles dans les fichiers twig suivants : tables.html.twig | user.html.twig | profil.html.twig | history.html.twig ainsi que dans le panel d'administration tableadmin.html.twig

2.2.2.3.1 Contenu

— Le contenu commence par du code CSS inline pour personnaliser le style de la page.

— En-tête : Il contient une image décorative de fond et un message Flash inactif inclus à partir du fichier `platform/shared/flashMessage.html.twig`.

— Formulaire de recherche : Un formulaire est affiché pour filtrer les tables en fonction du statut, de la disponibilité, de la date de création et du jeu.

À l'intérieur du formulaire, une div avec la classe "my-custom-class-for-errors" est utilisée pour afficher les éventuelles erreurs de validation du formulaire.

Ensuite, les éléments du formulaire sont disposés en utilisant des classes de mise en page flexbox de Bootstrap (`flex` et `items-start`) pour aligner les éléments en haut de leur conteneur.

Chaque élément du formulaire (statut, disponibilité, date de création, jeu) est encapsulé dans une div avec la classe "w-full md :w-1/4 pr-2", qui définit la largeur de l'élément en fonction de la largeur

de la fenêtre et spécifie un espacement à droite de l'élément.

À l'intérieur de chaque div, une balise `<div>` contenant le label est utilisée pour afficher le libellé de l'élément du formulaire. Les libellés sont stylisés en noir et en gras.

— Tableau des tables : Un tableau est affiché pour présenter les informations sur chaque table, y compris le jeu associé, les informations sur la table, les joueurs présents et les actions disponibles.

2.2.2.3.2 Filtres

— Les éventuelles erreurs de validation du formulaire sont affichées à l'intérieur d'une `<div>` avec la classe "my-custom-class-for-errors".

— Les éléments du formulaire sont agencés avec des classes de mise en page flex (`flex` et `items-start`) pour les aligner en haut de leur conteneur.

— Chaque élément du formulaire (statut, disponibilité, date de création, jeu) est encapsulé dans une `<div>` avec une classe définissant sa largeur (`w-full md:w-1/4 pr-2`), adaptée à la taille de la fenêtre et spécifiant un espacement à droite.

1. **Ajouter une Nouvelle Section au Formulaire :** Pour intégrer une nouvelle section, insérez simplement une nouvelle balise `<div>` avec les classes appropriées pour la mise en page. Par exemple, pour ajouter une section de filtre de catégorie, insérez une nouvelle `<div>` après la dernière `<div>` du formulaire existant.
2. **Ajouter un Nouveau Champ de Formulaire :** Utilisez les fonctions `form_label` et `form_widget` pour ajouter un nouveau champ de formulaire. Spécifiez l'identifiant, le libellé et les autres attributs nécessaires dans les options de ces fonctions. Par exemple, pour inclure un champ de filtre de catégorie, ajoutez le code suivant à la nouvelle section du formulaire :

```
<div class="w-full md:w-1/4 pr-2">
        {{ form_label(searchboard.category, 'Catégorie', {'label_attr':
        {'class': 'search-label'}}) }}
        {{ form_widget(searchboard.category, {'id': 'categorySelect',
        'placeholder': 'Choisir la catégorie'}) }}
```

3. **Personnaliser les Styles :** Au besoin, personnalisez les styles des nouveaux éléments en ajoutant des classes CSS supplémentaires ou en modifiant les classes existantes dans le code HTML.
4. **Gérer la Logique Côté Serveur :** Assurez-vous de mettre à jour la logique côté serveur pour traiter les données soumises par les nouveaux champs de formulaire, si nécessaire. Cela peut impliquer de mettre à jour les contrôleurs Symfony correspondants pour traiter les nouvelles données correctement.

Script JavaScript pour le formulaire de recherche :

Ce script JavaScript est utilisé pour rendre le formulaire de recherche interactif. Lorsqu'un utilisateur sélectionne une option dans l'un des menus déroulants du formulaire, le script soumet automatiquement le formulaire, déclenchant ainsi une nouvelle requête avec les paramètres de recherche mis à jour.

Voici comment le script fonctionne :

— La fonction `submitSearchForm()` est définie pour soumettre automatiquement le formulaire lorsqu'une sélection change. Elle est appelée à chaque fois qu'un événement de changement est détecté dans l'un des menus déroulants du formulaire.

— Des écouteurs d'événements sont ajoutés à chaque menu déroulant du formulaire :
  — L'écouteur d'événements pour l'élément avec l'id `statusSelect` est ajouté pour détecter les changements dans le statut de la table.
  — L'écouteur d'événements pour l'élément avec l'id `availabilitySelect` est ajouté pour détecter les changements dans la disponibilité de la table.
  — L'écouteur d'événements pour l'élément avec l'id `datecreationSelect` est ajouté pour détecter les changements dans la date de création de la table.
  — L'écouteur d'événements pour l'élément avec l'id `gameSelect` est ajouté pour détecter les changements dans le jeu associé à la table.

— Chaque fois qu'un changement est détecté dans l'un des menus déroulants, la fonction `submitSearchForm()` est appelée, ce qui déclenche la soumission automatique du formulaire.

2.2.2.3.3 Actions sur les Tables

Les actions suivantes sont disponibles sur chaque table dans le tableau :

— **Rejoindre la table** : Permet à un utilisateur de rejoindre une table ouverte avec des places disponibles. Exemple de code pour ajouter un bouton :

```
<a href="{{ path('app_join_board', {'id': board.id}) }}"
   class="text-white bg-primary hover:bg-blue-400
   focus:ring-4 focus:outline-none focus:ring-blue-300
   font-semibold rounded-lg text-sm px-5 py-2.5
   text-center me-2 mb-2">
    Rejoindre
</a>
```

— **Visionner la table** : Permet à un utilisateur de visionner une table en cours.

— **Reprendre la partie** : Permet à un utilisateur de reprendre une partie en cours à laquelle il participe déjà.

— **Quitter la table** : Permet à un utilisateur de quitter une table à laquelle il participe déjà.

Jeux	Informations	Joueurs	Action
`<td>` contenant les informations sur les jeux	`<td>` contenant les détails de la table	`<td>` contenant les avatars des joueurs	`<td>` contenant les boutons d'action

TABLE 2 – Liste des tables

— **Rejoindre la table** : L'utilisateur peut rejoindre une table en cliquant sur le bouton "Rejoindre". Le bouton est un lien HTML généré à partir de la fonction path() de Symfony, qui redirige l'utilisateur vers la route 'app_join_board' avec l'identifiant unique de la table (`board.id`).

— **Visionner la table** : Si la table est en cours, l'utilisateur peut visionner la table en cliquant sur ce lien. Il est redirigé vers une vue détaillée de la table en cours.

— **Reprendre la partie** : Si l'utilisateur est déjà dans la table et que la partie est en cours, il peut reprendre la partie en cliquant sur ce lien. Cela le redirige vers la page de reprise de la partie.

— **Quitter la table** : Si l'utilisateur est déjà dans la table, il peut la quitter en cliquant sur ce lien. Cela déclenche un processus pour le retirer de la table.

Les boutons d'action sont générés dynamiquement en fonction du statut de la table et de la participation de l'utilisateur. Les données des tables sont affichées dans un tableau avec des colonnes pour les jeux, les informations sur la table, les joueurs et les actions disponibles. Chaque cellule contient des balises HTML (`<td>`) pour afficher les données spécifiques.

2.2.2.3.4 Explication des boucles Twig et des filtres utilisés :

— **Boucle for :** La boucle `{% for board in boards %}` itère sur chaque élément (table) dans la liste des tables (`boards`).

— **Filtres Twig :** Les filtres Twig, tels que `map()`, `filter()`, et `length()`, sont utilisés pour manipuler les tableaux d'utilisateurs et pour effectuer des opérations telles que le calcul de la longueur du tableau ou la sélection d'éléments spécifiques.

— **Expressions conditionnelles :** Les expressions conditionnelles `{% if ... %}` sont utilisées pour vérifier certaines conditions et contrôler le flux d'exécution du code en fonction de ces conditions.

— **Conditions sur les tables :**
  — La boucle `{% for board in boards %}` itère sur chaque table (`board`) récupérée dans le contexte.
  — Les conditions suivantes sont appliquées pour chaque table :
    — `{% if (board.getNbAvailbleSlots() > 0 and board.status != 'FINISHED') or (board.getNbAvailbleSlots() == 0 and board.status == 'IN_GAME') %}` : vérifient si la table est ouverte et s'il y a des places disponibles ou si elle est en cours de jeu.

Ces conditions permettent de filtrer les tables à afficher en fonction de leur disponibilité et de leur statut.

2.2.2.4 games

2.2.2.4.1 Formulaire de création de table (boardRegister.html.twig)

Ce code HTML représente une page de création de table de jeu. Voici les différentes parties du code et leur contenu :

2.2.2.5 Contenu

1. **Section d'affichage des informations sur le jeu**
   — Cette section inclut une image du jeu, son nom, et potentiellement d'autres détails.
   — Les informations sont affichées dans une carte stylisée avec un arrière-plan flouté.
2. **Formulaire de sélection du nombre de joueurs**
   — Ce formulaire permet à l'utilisateur de sélectionner le nombre de joueurs pour la table de jeu.
   — Il affiche également le nombre minimum et maximum de joueurs autorisés pour le jeu sélectionné.
3. **Formulaire d'invitation des contacts**
   — Ce formulaire permet à l'utilisateur de sélectionner des contacts à inviter à la table de jeu.
   — Il comprend une option pour filtrer les contacts uniquement.
   — La liste de contacts est affichée, avec la possibilité de rechercher des contacts par nom.
   — Chaque contact est représenté par un élément de liste avec son nom et une image.
4. **Formulaire de définition du temps d'inactivité**
   — Ce formulaire permet à l'utilisateur de définir le temps maximum d'inactivité avant qu'un joueur ne soit expulsé de la table de jeu.
   — Il affiche également une description du fonctionnement de ce paramètre.
5. **Boutons de contrôle**
   — Des boutons sont fournis pour permettre à l'utilisateur de retourner en arrière ou de créer la table.

2.2.2.5.1 JavaScript

— Le script commence par récupérer des éléments HTML pertinents tels que les champs de formulaire et les listes de contacts.
— Il initialise les cases à cocher et les étiquettes associées pour la sélection des contacts à inviter.
— Il y a des fonctions pour réinitialiser les choix d'invitations lorsque le nombre de joueurs change, ainsi que pour limiter le nombre de contacts pouvant être invités en fonction du nombre de joueurs.
— Une fonction de filtrage est incluse pour permettre à l'utilisateur de rechercher des contacts par nom dans la liste.
— Le script gère également l'affichage d'un message lorsque aucun contact correspondant à la recherche n'est trouvé.

2.2.2.6 Page de Profil

La page de profil affiche les détails du profil d'utilisateur, y compris les informations personnelles, les jeux préférés, les tables en cours et l'historique des tables. Voici sa structure et son contenu :

2.2.2.6.1 Contenu

1. **Section Informations Utilisateur**
   — Cette section affiche les informations de base sur l'utilisateur telles que son nom d'utilisateur, sa description, sa date d'inscription et son statut (actif ou banni).
   — Elle contient également un bouton pour ajouter ou supprimer l'utilisateur en tant que contact (si ce n'est pas le profil de l'utilisateur actuel).
   — Un bouton est également présent, pour supprimer le compte de l'utilisateur. Il est visible uniquement pour le profil de l'utilisateur connecté.
2. **Section Jeux Préférés**
   — Cette section affiche une liste des jeux préférés de l'utilisateur.
   — Chaque jeu est représenté par une image miniature avec un lien vers la description du jeu.

3. **Section Contacts**
   — Cette section est visible uniquement pour l'utilisateur connecté.
   — Elle affiche une liste de contacts de l'utilisateur, chaque contact étant représenté par son nom d'utilisateur et une image d'avatar.
   — Chaque nom d'utilisateur est un lien vers le profil du contact.
4. **Section Tables en Cours**
   — Cette section affiche les tables de jeu en cours auxquelles l'utilisateur participe.
   — Chaque table est représentée par une rangée dans un tableau.
   — Les détails de chaque table incluent le jeu en cours, le nombre de places disponibles, le statut de la table (en attente, en cours, terminé), l'accessibilité de la table (ouverte ou fermée) et la date de création.
   — Pour l'utilisateur connecté, des boutons d'action sont disponibles, tels que "Reprendre" pour les tables en cours et "Quitter" pour les tables en attente.
5. **Section Historique des Tables**
   — Cette section affiche l'historique des tables de jeu passées auxquelles l'utilisateur a participé.
   — Chaque table est présentée de manière similaire à la section des tables en cours, avec des détails sur le jeu, les joueurs et l'action possible (par exemple, "Visionner").

2.2.2.6.2 JavaScript

Un script JavaScript est inclus pour afficher une boîte de dialogue de confirmation avant de supprimer un compte utilisateur.

2.2.3 home

2.2.3.1 La Page des jeux (games.html.twig)

Ce fichier Twig est responsable de l'affichage de la liste des jeux sur AGORA.

2.2.3.1.1 Structure

— Le fichier étend le modèle de base défini dans `platform/base.html.twig`.
— Le titre de la page est défini comme "Jeux".
— Le contenu principal est enveloppé dans un bloc `body`.

2.2.3.1.2 Contenu

— Le contenu est enveloppé dans un conteneur principal avec des classes de mise en page Tailwind CSS.
— Une image de fond décorative est ajoutée pour le style.
— Une section de navigation est ajoutée pour permettre la pagination des jeux.

2.2.3.1.3 Script JavaScript

Un script JavaScript est inclus pour gérer les actions de favoris. Il permet de basculer l'état de favori d'un jeu et de soumettre le formulaire associé pour enregistrer les changements.

2.2.3.1.4 Ajout de fonctionnalités et de boutons

Pour améliorer la page des jeux, vous pouvez ajouter de nouvelles fonctionnalités ou des boutons supplémentaires en suivant ces étapes :

1. Identifiez l'emplacement approprié dans le code pour ajouter votre fonctionnalité ou vos boutons, en veillant à respecter la structure existante.
2. Pour l'ajout de fonctionnalités, intégrez le code HTML, CSS et JavaScript nécessaires à la nouvelle fonctionnalité.
3. Pour l'ajout de boutons, localisez l'emplacement dans le fichier `dashboard/home/games.html.twig` où vous souhaitez insérer les boutons et ajoutez les balises HTML appropriées. Par exemple, pour inclure un bouton "Créer une partie", vous pouvez utiliser :

```
<a href="/dashboard/create-game" class="btn btn-primary">Créer une partie</a>
```

Veillez à personnaliser les styles des boutons selon vos besoins.

4. Testez la fonctionnalité ou les boutons nouvellement ajoutés pour vous assurer qu'ils fonctionnent correctement et qu'ils s'intègrent harmonieusement dans la mise en page existante, sans perturber l'expérience utilisateur.

2.2.3.1.5 Pagination des jeux

La pagination est un élément crucial pour améliorer l'expérience utilisateur lors de la navigation dans une liste de jeux. Voici comment vous pouvez implémenter la pagination dans votre page de jeux :

1. Identifier l'emplacement où vous souhaitez afficher les liens de pagination dans le fichier `dashboard/home/games.html.t`
2. Utiliser les balises HTML appropriées pour créer des liens vers les différentes pages de jeux. Par exemple :

```
<div class="pagination">
    <a href="/dashboard/games?page=1">1</a>
    <a href="/dashboard/games?page=2">2</a>
    <a href="/dashboard/games?page=3">3</a>
</div>
```

3. Adapter la logique de pagination côté serveur pour récupérer les jeux en fonction de la page demandée.
4. Tester la pagination pour vous assurer que les liens fonctionnent correctement et que la navigation entre les pages est fluide.

2.2.3.1.6 Lien Back-end et Front-end

La liaison entre le back-end Symfony et le front-end est essentielle pour assurer le bon fonctionnement des fonctionnalités interactives de la page des jeux. Voici comment vous pouvez assurer cette liaison :

1. Soumission du formulaire favori : Le bouton favori sur chaque jeu déclenche une soumission de formulaire pour mettre à jour l'état favori du jeu pour l'utilisateur connecté. Voici comment cela fonctionne :
   (a) Lorsque l'utilisateur clique sur le bouton étoile, un événement JavaScript est déclenché pour inverser l'état de favori du jeu.
   (b) Le formulaire associé est soumis de manière asynchrone à l'aide de JavaScript, avec l'état de favori mis à jour comme paramètre.
   (c) Côté back-end, le contrôleur Symfony associé à la route `app_game_favorite` est appelé pour mettre à jour l'état favori du jeu dans la base de données.
2. Gestion de la pagination : La pagination est également une fonctionnalité importante pour améliorer l'expérience utilisateur lors de la navigation dans une grande liste de jeux. Voici comment elle est mise en œuvre :
   (a) Les liens de pagination sont générés côté front-end avec des balises HTML appropriées pour permettre à l'utilisateur de naviguer entre les différentes pages de jeux.
   (b) Ces liens sont associés à des URL spécifiques qui incluent des paramètres de requête, tels que `page=1`, `page=2`, etc.
   (c) Côté back-end, Symfony récupère les jeux en fonction de la page demandée à l'aide de la logique de pagination définie dans le contrôleur correspondant.
3. Test et validation : Il est crucial de tester ces fonctionnalités pour garantir qu'elles fonctionnent correctement et offrent une expérience utilisateur fluide. Assurez-vous de tester à la fois la soumission du formulaire favori et la navigation via la pagination pour identifier et résoudre tout problème éventuel.

2.2.3.2 La Page de Description des Jeux (`games.html.twig`)

Ce fichier Twig est responsable de l'affichage des descriptions détaillées des jeux sur AGORA.

2.2.3.2.1 Structure

— Le fichier étend le modèle de base défini dans `platform/base.html.twig`.
— Le titre de la page est dynamiquement défini en fonction du nom du jeu.
— Le contenu principal est enveloppé dans un bloc `body`.

2.2.3.2.2 Contenu

— Le contenu est affiché dans un conteneur principal avec des classes de mise en page Tailwind CSS pour assurer une présentation esthétique.
— L'image du jeu est affichée en arrière-plan avec une superposition floue pour une meilleure esthétique visuelle.
— Les détails du jeu, tels que le nom et la description, sont affichés de manière claire et lisible.

2.2.3.2.3 Exemples d'ajouts de boutons et de fonctionnalités

Pour améliorer la page de description des jeux, vous pouvez ajouter des boutons et des fonctionnalités supplémentaires :

1. **Ajouter un Bouton de Retour** : Intégrez un bouton permettant à l'utilisateur de revenir à la liste des jeux. Utilisez le code suivant pour ajouter ce bouton :

```
<a href="/dashboard/games" class="btn btn-primary">
    Retour à la Liste des Jeux
 </a>
```

2. **Modifier le Style du Titre** : Personnalisez le style du titre du jeu en utilisant des classes CSS de Tailwind. Par exemple, pour mettre en évidence le titre, vous pouvez utiliser :

```
<h1 class="text-3xl font-bold text-gray-800 mb-4">
    {{ game.name }}
</h1>
```

3. **Ajouter des Liens Supplémentaires** : Pour fournir des ressources supplémentaires aux utilisateurs, vous pouvez ajouter des liens vers des guides de stratégie ou des vidéos de tutoriels. Voici un exemple :

```
<a href="{{ game.strategyGuideURL }}" class="text-blue-500 hover:underline
 Guide de Stratégie
</a>
```

4. **Ajouter des Informations Supplémentaires** : Enrichissez la description du jeu en ajoutant des détails tels que le nombre de joueurs recommandé, la durée moyenne de jeu et les composants inclus. Voici un exemple :

```
<p class="text-gray-600">
     <strong>Nombre de Joueurs :</strong> {{ game.playerCount }}
</p>
<p class="text-gray-600">
    <strong>Durée de Jeu :</strong> {{ game.playingTime }}
</p>
<p class="text-gray-600">
    <strong>Composants :</strong> {{ game.components }}
</p>
```

5. **Intégration de Tailwind CSS** : Utilisez les classes de Tailwind CSS pour styliser les éléments de la page et créer une mise en page réactive.

2.2.3.2.4 Script JavaScript

Pour ajouter des fonctionnalités interactives, telles que l'ajout aux favoris ou le partage sur les réseaux sociaux, vous pouvez inclure des scripts JavaScript personnalisés. Voici un exemple de code avec une fonction JavaScript :

```
<button id="favoriteButton" class="btn btn-primary" onclick="toggleFavorite()">
    Ajouter aux Favoris
</button>

<script>
    function toggleFavorite() {
    // Logique pour basculer l'état de favori du jeu
     // et mettre à jour l'interface utilisateur
}
</script>
```

2.2.4 publisher

2.2.5 registration

2.2.5.1 Page d'inscription (register.html.twig)

Cette page est une page d'inscription qui permet aux utilisateurs de créer un nouveau compte sur la plateforme.

2.2.5.1.1 Contenu

1. **Formulaire d'inscription** : Le formulaire permet à l'utilisateur de saisir un identifiant, un mail et un mot de passe pour s'inscrire à l'application. Il comprend :
   — **Champ "Identifiant"** : Permet à l'utilisateur de saisir un identifiant.
   — **Champ "Email"** : Permet à l'utilisateur de saisir son adresse Émail.
   — **Champ "Mot de passe"** : Permet à l'utilisateur de saisir un mot de passe.
   — **Bouton "Créer mon compte"** : Permet à l'utilisateur de soumettre le formulaire.
   — **Lien "Se connecter"** : Rediriger les utilisateurs vers la page de connexion pour pouvoir se connecter s'il possède déjà un compte.
2. **Messages d'erreur** : Si des erreurs sont détectées lors de la soumission du formulaire, elles sont affichées dans une section d'alerte rouge. Chaque erreur est affichée sous forme d'une liste à puces..
3. **Arrière-plan décoratif** : Deux éléments décoratifs en forme de gradient sont positionnés en arrière-plan pour ajouter une esthétique visuelle à la page.

2.2.6 security

2.2.6.1 Page de Connexion (login.html.twig)

Ce template est utilisé pour afficher le formulaire de connexion de l'application.

2.2.6.1.1 Contenu

1. **Formulaire de connexion** : Le formulaire permet à l'utilisateur de saisir son identifiant et son mot de passe pour se connecter à l'application. Il comprend :
   — **Champ "Identifiant"** : Permet à l'utilisateur de saisir son identifiant.
   — **Champ "Mot de passe"** : Permet à l'utilisateur de saisir son mot de passe. Un lien est également fourni pour permettre à l'utilisateur de réinitialiser son mot de passe en cas d'oubli.

— **Bouton "Se connecter"** : Permet à l'utilisateur de soumettre le formulaire.
— **Lien "Créer un compte"** : Redirige l'utilisateur vers la page de création de compte s'il n'a pas encore de compte.

2. **Messages Flash** : Les messages flash sont utilisés pour afficher des notifications à l'utilisateur, par exemple pour indiquer si la connexion a réussi ou échoué. Les messages sont affichés en fonction de leur type (succès, avertissement, erreur) et sont associés à des classes de couleur correspondantes.
3. **Arrière-plan décoratif** : Deux éléments décoratifs en forme de gradient sont positionnés en arrière-plan pour ajouter une esthétique visuelle à la page.

2.2.7 shared

2.2.7.1 Gestion des messages flash (flashMessage.html.twig)

2.2.7.1.1 Contenu

Ce code affiche les messages flash sous forme de notifications colorées. Les notifications peuvent être de types 'success', 'warning', 'error', ou 'primary'. Chaque type de notification est stylisé différemment.

— **Success (success)** : Ce type de message est utilisé pour indiquer à l'utilisateur qu'une opération s'est déroulée avec succès, comme la soumission réussie d'un formulaire.
— **Warning (warning)** : Ce type de message est utilisé pour avertir l'utilisateur d'une condition qui nécessite son attention, mais qui n'est pas nécessairement une erreur.
— **Error (error)** : Ce type de message est utilisé pour informer l'utilisateur qu'une erreur s'est produite, comme une saisie incorrecte dans un formulaire.
— **Primary (primary)** : Ce type de message peut être utilisé pour fournir des informations importantes à l'utilisateur, sans être nécessairement un succès, un avertissement ou une erreur.

Si un message flash n'est ni un 'success', ni 'warning, ni 'error', il est automatiquement considéré comme 'primary'.

2.2.7.1.2 JavaScript

Lorsqu'un bouton est cliqué, la fonction dismissFlash() est appelée. Cette fonction trouve le message flash parent du bouton cliqué et le supprime du DOM, permettant ainsi à l'utilisateur de fermer la notification.

2.2.7.1.3 Utilisation

Pour utiliser ce code dans d'autres pages, vous pouvez l'inclure dans vos autres pages Twig à l'aide de l'instruction :

```
{% include 'platform/shared/flashMessage.html.twig' %}
```

Cette approche permet de réutiliser facilement le code des messages flash sur plusieurs pages sans avoir à le répéter à chaque fois. Cela favorise également la maintenance du code en regroupant les fonctionnalités liées dans des fichiers distincts.

2.2.8 users

2.2.8.1 Page paramètres de compte (editProfileTemplate.html.twig)

2.2.8.1.1 Contenu

Le code comprend les éléments suivants :

1. **Structure de base HTML :** Le code commence par une structure HTML de base, avec des balises `<div>` pour la mise en page et l'organisation du contenu.
2. **Formulaire de Modification :** Un formulaire est créé à l'aide de la fonction `form_start()` de Twig. Les champs du formulaire comprennent l'identifiant de l'utilisateur, son adresse e-mail et deux champs de mot de passe pour le nouveau mot de passe et sa confirmation.

3. **Personnalisation des champs :** Chaque champ de formulaire est personnalisé avec des attributs CSS pour le style, tels que la couleur du texte, les ombres, les bordures, etc.
4. **Bouton de Soumission :** Un bouton de soumission est inclus à l'aide de la fonction `form_row()` de Twig. Ce bouton est également stylisé avec des attributs CSS pour correspondre au thème de l'interface utilisateur.
5. **Gestion des Routes :** Selon le chemin d'accès actuel de l'application, le formulaire est configuré pour envoyer les données à différentes routes. Cela est réalisé à l'aide de conditions Twig intégrées dans les attributs du formulaire.

2.2.8.1.2 Ajout de fonctionnalités

— Pour ajouter un champ de texte :

```
<div>
    {{ form_label(form.nom, 'Nom', {'label_attr': {'class': 'block text-sm font-medium leading
    <div class="mt-2">
        {{ form_widget(form.nom, {'attr': {'class': 'block w-full rounded-md border-0 py-1.5 t
    </div>
</div}
```

— Pour ajouter un champ de sélection :

```
<div>
    {{ form_label(form.pays, 'Pays', {'label_attr': {'class': 'block text-sm font-medium leadi
    <div class="mt-2">
        {{ form_widget(form.pays, {'attr': {'class': 'block w-full rounded-md border-0 py-1.5
    </div>
</div>
```

— Pour ajouter un bouton :

```
{{ form_row(form.envoyer, {'attr': {'class': 'flex w-full justify-center rounded-md bg-primary
```

2.2.8.2 Paramètres administrateur/utilisateur (editAdminProfile.html.twig, editUserProfile.html.twig)

2.2.8.2.1 Contenu

Les deux pages sont relativement identique, les deux contiennent une inclusion de la page editPofileTemplate.html.twig. Les deux diffère sur la page qu'ils étendes.

— editAdminProfile.html.twig : étend le layout administrateur. Permet d'afficher le dasboard administrateur.
— editUserProfile.html.twig : étend le layout utilisateur. Permet d'afficher le dasboard utilisateur.