Rapport de Remise Projet Othello (Client/Server/Common)

Auteur : Alec Waumans _____ ID : 58399 _____

Date : 11/ 09/ 202514 août 2025

Rapport du project d'atlir Othello Client-Server

Trois modules importables (Client / Serveur / Commun)

Les trois modules Maven sont présents et respectent la nomenclature demandée :

— `OthelloClientAlec`

— `OthelloServerAlec`

— `OthelloCommonAlec`

Le module serveur concentre la logique métier et le modèle du jeu. Il implémente les règles (contrôleurs et stratégies), la gestion d'état (Board, Game, State, etc.) et l'accès aux données via SQLite (script `othello.sql`, dépendance sqlite-jdbc). La base est initialisée par `othello.sql` (création des 4 tables, données de démonstration et alignement de la séquence `'user'` sur `MAX(USER.id)`). Le serveur est l'unique source de vérité applicative : le client n'a aucun accès direct au modèle et ne peut pas l'altérer. Son `pom.xml` ne déclare qu'une dépendance vers **OthelloCommonAlec** (pas vers le client), garantissant l'encapsulation du backend et la cohérence du jeu.

Le module client est dédié à l'interface et à l'expérience utilisateur. Il s'appuie sur JavaFX/FXML (LoginView.fxml, vues et panneaux) et des contrôleurs de présentation qui affichent l'état reçu du serveur et émettent des actions utilisateur, sans logique métier locale. Son `pom.xml` ne référence que **OthelloCommonAlec** et les bibliothèques JavaFX, sans dépendre du serveur ; le client peut ainsi s'exécuter sur une machine distante, avec une communication strictement bornée par les messages définis dans le module commun.

Le module *common* définit le contrat d'échange entre client et serveur. Il regroupe les objets sérialisables et DTO (par ex. `Message`, `Type`, `MessageProfile`, `MessagePlay`, `MessageInit`, `MessageGameInfo`, `PositionDTO`, `User`, `Members`, `Config`, `GameInfo`) qui structurent le protocole applicatif. L'état de la partie est exposé via `GameInfo.GameStatus` (énumération côté commun) et est mappé depuis l'état interne du serveur. Le protocole comprend les types `PROFILE`, `PLAY`, `UNDO`, `REDO`, `QUIT`, `GAMEINFO`, `INIT` et `CONFIG`. Le fait que serveur et client dépendent de *common*, sans jamais dépendre l'un de l'autre, garantit la compatibilité binaire, la stabilité du protocole et l'indépendance de déploiement.

J'ai repris comme base le laboratoire client–serveur et conservé des traces console côté serveur et côté client pour visualiser les messages échangés et faciliter le débogage. J'ai ensuite intégré l'interface FXML ainsi que JDBC, puis finalisé en incorporant le modèle du jeu dans le module serveur et l'interface JavaFX côté client. J'ai ajouté un type de message supplémentaire nommé `CONFIG` (taille du plateau, mode de jeu) afin de séparer clairement les paramètres d'initialisation du reste du protocole et de simplifier l'initialisation côté client.

Conformité aux exigences de la remise

Exigence	Statut	Observations
Trois projets NetBeans/Maven importables (`OthelloClientAlec`, `OthelloServerAlec`, `OthelloCommonAlec`)	**OK**	—
Architecture client/serveur avec module commun	**OK**	Dépendances : *client* ↔ *common* ; *serveur* ↔ *common*.
Types de messages `PROFILE`, `PLAY`, `UNDO`, `REDO`, `QUIT`, `GAMEINFO`, `INIT`, `CONFIG`	**OK**	`PASS` non requis.
Type `PASS` (optionnel)	**Non**	Non implémenté (optionnel, non exigé).
État de partie exposé via `GameInfo.GameStatus` ; objets `User`, `PositionDTO`, `GameInfo`	**OK**	Écart assumé par rapport à un `GameState` global dans *common*.
Interface FXML de connexion avec *email*/*host*/*port*	**OK**	Valeurs par défaut : `anonyme@esi.be`, `localhost`, 12345.
Persistance JDBC en couches ; tables `SEQUENCE`, `USER`, `GAME`, `SCORE`	**OK**	DAO + `DBManager` ; clés étrangères activées.
Un seul fichier SQL de création dans les ressources du serveur	**OK**	`sqliteDB/othello.sql` (création + démo).
Séquence `'user'` alignée sur `MAX(USER.id)`	**OK**	Mise à jour de `SEQUENCE` après inserts de démo.
Attribution de l'ID utilisateur via séquence si email inconnu ; récupération sinon	**OK**	Normalisation d'e-mail non implémentée (risque mineur de doublons).
Envoi d'un `MessageProfile` et affichage email/ID à réception	**OK**	Affichage dans la *MainPane* à la réception de `PROFILE`.

Fonctionnalités : description détaillée

1. Transformation en application Serveur–Clients

Ce qui a été fait. Le serveur héberge l'intégralité du modèle de jeu et de la logique d'évolution (`server.model.*`), expose une API *message-orientée* et gère la persistance SQLite (script `sqliteDB/othello.sql`, dépendance `sqlite-jdbc`). Le script crée les 4 tables et aligne la séquence `'user'` sur `MAX(USER.id)`. Le client se limite à afficher l'état reçu et à transmettre des intentions (`PROFILE`, `PLAY`, `UNDO`, `REDO`, `QUIT`, `INIT`, `CONFIG`, `GAMEINFO`). Le contrat (types, messages, DTO) est défini dans `OthelloCommonAlec`.

Ce qui n'a pas été fait. Pas de message `PASS` (optionnel). La gestion des *members* n'est pas développée comme au laboratoire : la structure est conservée (réception via `PROFILE`), mais il n'y a pas de vue dédiée ni de fonctionnalités avancées ; elle pourra être complétée ultérieurement.

Comment tester.

1. Lancer le serveur (voir section *Build & Exécution*).
2. Démarrer un ou plusieurs clients, saisir *email*, *host*, *port*, puis se connecter.
3. Vérifier la réception de `PROFILE` côté client (identité et, le cas échéant, liste des membres minimale).
4. Jouer quelques coups (`PLAY`), puis tester `UNDO`/`REDO`.
5. Quitter proprement via le *bouton* `QUIT` (barre de menu). Les consoles client/serveur affichent les messages échangés, ce qui facilite le diagnostic.

Bugs connus. Aucune régression bloquante observée lors des tests manuels. Un cas historique de *threads* résiduels après fermeture a été corrigé, mais la séquence de fermeture mérite encore d'être durcie. En fin de partie, certains contrôles UI peuvent rester actifs : il est parfois possible de *continuer* à jouer malgré l'état `ENDGAME`.

Limitations. Pas de mécanisme de reconnexion automatique. En fin de partie, l'UI affiche correctement la fin du jeu et le *GameStatus*, mais il n'y a pas encore de redirection vers une nouvelle partie ni de tableau de scores (exploitable via la base de données). À ce stade, le client se ferme via `QUIT` ou par fermeture de la fenêtre.

2. Interface graphique FXML de connexion

Ce qui a été fait. `LoginView.fxml` définit trois champs : *email* (valeur par défaut `anonyme@esi.be`), *host* (`localhost`) et *port* (`12345`), ainsi qu'un bouton *Connect*. Le contrôleur `LoginController` instancie le client, initie la connexion au serveur et, en cas de succès, bascule vers l'écran de configuration du jeu.

Ce qui n'a pas été fait. Pas de validation avancée côté client (format d'email, normalisation `trim`/`toLowerCase`, contrôle strict de la plage de ports). Pas de persistance des derniers paramètres saisis.

Comment tester. Lancer le client, conserver les valeurs par défaut ou saisir un email, un hôte et un port, puis cliquer sur *Connect*. En cas de connexion réussie, l'interface passe à l'écran de configuration. En cas d'hôte/port incorrects, un message d'erreur s'affiche dans la console et/ou via une alerte.

Bugs connus. Si le port est invalide ou si l'hôte est injoignable, le message d'erreur peut rester succinct.

Limitations. Absence de filtrage/validation des entrées dans le formulaire (email non normalisé, port non vérifié finement) et pas de mémorisation des champs entre deux exécutions.

3. Persistance JDBC en couches (SQLite)

Ce qui a été fait. Les DAO (`UserDAO`, `GameDAO`, `ScoreDAO`) encapsulent l'accès à la base ; `DBManager` centralise la connexion, active les clés étrangères et gère la `SEQUENCE`. Un **seul** script, `sqliteDB/othello.sql`, est embarqué côté serveur : il crée les tables `SEQUENCE`, `USER`, `GAME`, `SCORE`, insère des *données de démonstration* (utilisateurs, parties, scores) et **aligne la séquence** `'user'` sur `MAX(USER.id)`. La méthode `UserDAO.findOrCreate(email)` récupère l'ID existant ou insère l'utilisateur avec un nouvel ID provenant de `DBManager.getNextId("user")` (transaction courte).

Comment tester.

1. Supprimer la base `othello.db` si présente, puis lancer le serveur : le script `othello.sql` recrée la structure, les données de démo et aligne la séquence.
2. Lancer un client, se connecter avec un email nouveau : vérifier l'insertion dans `USER` et l'ID issu de la séquence.
3. Démarrer une partie, jouer quelques coups : contrôler les insertions dans `GAME` et `SCORE`.
4. Reconnecter le *même* email : l'ID retourné doit rester identique.

Limitations. Pas de verrouillage pessimiste ; la concurrence est limitée par le serveur mais des sections `synchronized` autour de `findOrCreate` et des insertions critiques renforceraient l'atomicité. Pas de nettoyage automatique des parties abandonnées/incomplètes. Pas de normalisation avancée des emails (trim, `toLowerCase`, suppression des espaces invisibles) : risque mineur de doublons « quasi identiques ». Chemin de base par défaut adapté au poste de développement ; pour un déploiement autonome, prévoir un répertoire d'exécution dédié.

4. Attribution de l'ID utilisateur par email

Ce qui a été fait. La méthode `UserDAO.findOrCreate(email)` gère la récupération ou la création d'un utilisateur à partir de son email :

— Recherche via `SELECT id FROM USER WHERE email = ?`.

— Si l'email existe, l'ID correspondant est retourné.

— Si l'email est absent, la `SEQUENCE` fournit le prochain identifiant, puis insertion via `INSERT INTO USER(id, email) VALUES (?, ?)`.

La valeur de `SEQUENCE('user')` est alignée au démarrage par le script SQL afin d'assurer la continuité des identifiants.

Ce qui n'a pas été fait. Aucune normalisation avancée des adresses email :

— La casse (majuscules/minuscules) n'est pas harmonisée.

— Les espaces ou caractères superflus ne sont pas systématiquement supprimés.

Comment tester.

1. Se connecter deux fois avec exactement le même email : l'ID retourné doit rester identique.
2. Se connecter avec un nouvel email : un nouvel ID doit être généré.

Bugs connus. En cas de **connexions simultanées** avec la même adresse email, un client peut rester non connecté ou rencontrer un conflit transitoire. Il faudrait **encadrer** la création/-recherche utilisateur dans des blocs `synchronized` côté serveur, afin qu'une seule connexion à la fois puisse créer ou récupérer l'ID d'un email donné et éviter toute lecture/écriture concurrente non maîtrisée sur la séquence et la table `USER`.

Limitations.

— Contrainte d'unicité uniquement sur la chaîne exacte de l'email.

— Risque de collision si des variantes d'un même email sont utilisées (`A@B` vs `a@b`, présence d'espaces, etc.).

— Une normalisation préalable des adresses (trim, `toLowerCase`, suppression de caractères invisibles) pourrait réduire ces risques.

5. Message `PROFILE` et affichage côté client

Ce qui a été fait. Lors de la connexion d'un client, le serveur envoie un `MessageProfile` contenant les informations de l'utilisateur (ID et email). Côté client :

— `OthelloClient.handleMessageFromServer` intercepte le message de type `PROFILE`.

— Les attributs `mySelf` et `members` sont mis à jour avec les données reçues.

— L'interface met à jour la barre supérieure (*MainPane*) pour afficher l'ID et l'email de l'utilisateur connecté.

Ce qui n'a pas été fait.

— Pas de page de profil détaillée (photo/avatar, historique des parties, statistiques).

— Pas de personnalisation avancée de l'affichage des informations de profil.

Comment tester.

1. Lancer le serveur, puis un client.
2. Se connecter avec un email valide.
3. Vérifier que l'ID et l'email s'affichent correctement dans l'interface (en haut de la *MainPane*).

Bugs connus. Aucun bug critique constaté dans le traitement et l'affichage des informations de profil.

Limitations.

— Absence de notifications visuelles avancées (toasts, popups enrichis) pour signaler la réception ou la mise à jour du profil.

Build & Exécution (machine de l'école)

Prérequis (postes de l'école)

— **JDK 17** (ou supérieur) installé et sélectionné par défaut.

— **Maven 3.8+** disponible dans le `PATH` (`mvn -v` doit fonctionner).

Initialisation de la base de données

Au premier démarrage du serveur, le script `othello.sql` (fourni dans les ressources) crée les 4 tables `SEQUENCE`, `USER`, `GAME`, `SCORE`, **insère des données de démonstration** et **aligne la séquence `'user'`** sur `MAX(USER.id)`.

Aucune action manuelle n'est requise sur les postes de l'école. En cas de besoin de réinitialisation complète, supprimer le fichier `othello.db` généré par l'application, puis relancer le serveur pour ré-exécuter `othello.sql`.

Exécution depuis l'IDE (NetBeans/IntelliJ/Eclipse)

1. Ouvrir la racine du projet comme projet Maven.
2. Lancer le **serveur** (*Run* du module serveur). Vérifier dans la console que l'écoute sur le port (`12345` par défaut) est active.
3. Lancer le **client** (*Run* du module client). L'écran de connexion (FXML) propose 3 champs : *email* (défaut `anonyme@esi.be`), *host* (défaut `localhost`), *port* (défaut `12345`).

4. Saisir un email valide. Pour vérifier rapidement l'intégration base de données, on peut utiliser les e-mails de démo présents dans `othello.sql` : `demo1@esi.be` ou `demo2@esi.be`.
5. Après connexion, l'interface affiche l'ID et l'e-mail de l'utilisateur (message `PROFILE` renvoyé par le serveur).

Tests rapides de non-régression

1. **Connexion/Profil** : se connecter deux fois avec exactement le même e-mail ⇒ même ID.
2. **Nouvel utilisateur** : se connecter avec un e-mail inédit ⇒ création en base + ID issu de la séquence.
3. **Affichage UI** : ID et e-mail visibles côté client juste après la réception de `PROFILE`.
4. **Persistance** : démarrer une partie, jouer quelques coups ⇒ lignes insérées dans `GAME`/`SCORE`.

Dépannage (postes de l'école)

— **Port déjà utilisé** : le serveur refuse de démarrer ⇒ vérifier qu'aucun autre serveur ne tourne (`netstat`/`tasklist`) ou changer le port.

— **Accès BD** : si la base est verrouillée/corrompue, supprimer le fichier `othello.db` local puis relancer (le script `othello.sql` réinitialise).

— **JavaFX** : en cas d'erreur de lancement côté client depuis l'IDE, confirmer que le **JDK 17+** est sélectionné et que le projet utilise bien la configuration JavaFX/Maven prévue.

Tests manuels recommandés

1. Connexion par défaut (`anonyme@esi.be`, `localhost`, port serveur).
2. Connexions multiples avec des emails distincts ;
3. Déroulement d'une partie : `PLAY` suivi de `UNDO`/`REDO` ; inspection de la base (`USER`/`GAME`/`SCORE`).
4. Quitter via `QUIT` ;

Limites & pistes d'amélioration

— **Message PASS** non implémenté.

— **Chemin DB** relatif : paramétrer un emplacement configurable (`Config`) et générer la DB au premier lancement.

— **Robustesse réseau** : ajouter reconnexion et messages d'erreur plus explicites dans l'UI.

— **Tests** : compléter par des tests unitaires/integration pour les DAO et le protocole de messages.

— **UX** : enrichir l'UI (profil détaillé, scoreboard, notifications).

Conclusion

Dans l'ensemble, le programme est fonctionnel et les consignes du cahier des charges ont été respectées au mieux. Toutefois, certains problèmes subsistent et nécessiteraient d'être corrigés pour atteindre un niveau de fiabilité et de robustesse optimal. J'ai veillé à maintenir une architecture stable et conforme aux principes étudiés, même si certaines portions de code comportent des solutions temporaires ("pansements") pour répondre à des contraintes techniques ou de temps. Avec davantage de temps, plusieurs points pourraient être améliorés ou finalisés : optimisation du code, gestion plus fine de la concurrence, enrichissement des fonctionnalités et perfectionnement de l'interface utilisateur. Le manque de temps reste le principal facteur ayant limité la finalisation complète de certaines parties du projet. Malgré cela, le projet remplit ses objectifs principaux et constitue une base solide pour des évolutions futures.