CÉGEP DE RIVIÈRE-DU-LOUP

TECHNIQUES DE L'INFORMATIQUE — ÉCOLE DU WEB —

Programmation web avancée

Application mobile Android avec base de données transactionnelle

(Intra 2 - A2025)

Les enseignants se réservent le droit de modifier l'énoncé à tout moment pour assurer la cohérence avec le contenu du cours et les compétences visées.

Objectif général

Développer **individuellement** une application mobile Android (React Native / Expo) connectée à une **API transactionnelle** (MariaDB, Sql ou MySql selon vos choix), déployée sur votre **serveur VPS OVH**.

L'application doit permettre la gestion d'une flotte de navires pirates appartenant à un armateur.

Règles d'engagement

Travail individuel

- Ce travail est strictement individuel.

Collaboration et entraide

- Les discussions entre étudiants sont permises uniquement pour obtenir des conseils, échanger sur des approches ou clarifier la compréhension des consignes.
- Aucune aide directe (partage de code, capture d'écran, ou correction d'erreurs par un pair) n'est permise.
- Toute contribution directe au code par une autre personne est considérée comme de la triche.

Utilisation de l'intelligence artificielle (IA)

- Vous pouvez utiliser des outils d'IA à titre de soutien (par exemple pour clarifier une notion, proposer une structure ou générer un extrait très limité de code).
- Vous devez toutefois :
  - Citer vos sources (nom de l'outil et date d'utilisation).
  - Comprendre entièrement tout le code que vous intégrez.
  - Documenter chaque ligne ou bloc de code provenant d'une suggestion d'IA.
  - N'inclure que le code essentiel à la réalisation de votre projet.
  - Tout code fantôme (non utilisé, non compris, ou non justifié) sera considéré comme du code copié sans compréhension et sera supprimé ou pénalisé lors de la correction.

Aide de l'enseignant

- Vous pouvez poser des questions à votre enseignant, à condition de démontrer votre autonomie dans la démarche et de montrer que vous maîtrisez les notions normalement acquises. Présentez le problème clairement.
  - Montrez les étapes déjà tentées.
  - Identifiez ce que vous ne comprenez pas.
- Les questions d'ordre général (erreurs de logique, approche de conception, clarification d'exigences) sont encouragées.

Présence et engagement

- La présence en classe est obligatoire pendant les séances dédiées au travail long.
- Vous devez travailler activement en laboratoire et poursuivre le travail à la maison dans les périodes prévues aux devoirs.
- L'objectif est de démontrer une progression personnelle du projet, observable dans vos commits GitHub et vos itérations.

Partie 1 – Planification et organisation du développement

Découpage en itérations

- Lire attentivement les exigences fonctionnelles ci-dessous.
- Diviser le travail en itérations fonctionnelles **de 1 à 2 heures maximum** chacune.
- Chaque itération doit être testée et fonctionnelle (ne pas empiler du code non testé).
- Exemples d'itérations :
  - Mise en place du projet Expo / Node.js
  - Création du modèle Utilisateur
  - Création du modèle Navire
  - API de login/logout
  - Affichage de la liste des navires
  - Ajout / retrait d'équipage, etc.

Stratégie de branche unique

- Travailler sur **une seule branche principale (main)**.
- Chaque itération = un commit complet, testé et propre.
- Sur chaque *push*, exécuter les **tests Maestro** pour valider la fonctionnalité avant d'aller plus loin.
- Vous êtes responsables de vos **pull requests** et de la qualité du code soumis.

Développement coopératif

- Pendant le projet :
  - L'enseignant publiera des **billets (issues)** dans le dépôt principal.
  - Ces billets sont **prioritaires** et doivent être traités dès leur publication.

- Étapes à suivre :
  - Fork du dépôt principal.
  - Créer une branche spécifique pour le billet (feature-description-XXXX).
  - Implémenter la fonctionnalité ou la correction demandée.
  - Faire les tests locaux (Maestro…).
  - Faire un commit clair et descriptif.
  - Push vers votre dépôt (verifié par Maestro).
  - Créer un Pull Request vers le dépôt principal.
  - Ajouter votre enseignant comme "Reviewer".
  - Attendre la validation avant de merger.

Exigences fonctionnelles de l'application

- Authentification
  - Page de Login (nom d'utilisateur et mot de passe).
  - Possibilité de Logout (réinitialise la session / token).
- Liste des navires
  - Afficher uniquement les navires du port.
  - Mode d'affichage : table/liste ligne par ligne. Chaque ligne est cliquable et peut être sélectionnée. Permettre la sélection multiple (ex. appui long ou cases de sélection)..
  - Afficher les informations clés : nom du navire, port actuel, équipage, quantité d'or…
- Actions selon le rôle
  - a. Si l'utilisateur est un simple "Pirate" :
    - Sélectionner un navire et (type jeter par dessu bord):
      - Changer de port.
      - Recruter ou renvoyer des membres d'équipage *(un marin renvoyé, c'est un marin jeté à la mer !)*.
      - Ajouter ou retirer de l'or du trésor *(l'or retiré est jeté par-dessus bord, perdu à jamais)*.
  - b. Si l'utilisateur est un "Amiral_Pirate" :
    - Ajouter un nouveau navire (nom, type, port d'attache, équipage initial, or …).
    - Peut transférer de l'or d'un navire dans un autre.
    - S'il y a transfert d'or d'un bateau à un autre, c'est considéré comme du piratage! Donc on ajoute un champs qui dit combien de fois le navire a été pillé.

Base de données transactionnelle

- Utiliser une base de données relationnelle (ex. PostgreSQL, MySQL) avec gestion transactionnelle.
- Chaque modification critique (ajout/retrait d'or, équipage, destruction de navire) doit être enregistrée dans une transaction atomique.
- Implémenter des contraintes de validation (ex. ne pas avoir d'équipage négatif).

- Mettre en pratique les transactions atomiques pour des opérations critiques sur les navires dans la base de données.
- Toutes les opérations doivent être atomiques : si une étape échoue, aucune modification ne doit être appliquée.
- Fonctionnalité : Transfert d'or entre navires
  - Dans votre application, chaque navire pirate possède une quantité d'or.
  - Les pirates peuvent transférer de l'or d'un navire à un autre via l'API Node.js.
  - Les opérations doivent respecter l'intégrité des données même en présence de **concurrence**.
  - Implémenter un endpoint /transferGold qui effectue un transfert d'or atomique :
    - Toutes les étapes (décrément du navire source, incrément du navire destination) doivent se faire dans une seule transaction SQL.
    - Si une étape échoue, aucune modification ne doit être appliquée (ROLLBACK).
  - Simuler un délai dans la transaction (par ex. 3 à 5 secondes) pour représenter un transfert long.
    - Utilisez await new Promise(r => setTimeout(r, 3000)) côté Node.js après le verrouillage du navire source.
  - Gestion de la concurrence :
    - Si une autre transaction concurrente tente de modifier le même navire en même temps, les deux transactions doivent s'abandonner pour éviter toute incohérence.
    - L'idée est d'utiliser un verrou exclusif (SELECT ... FOR UPDATE ou équivalent) et détecter les conflits.

- Vérification des contraintes :
  - Le navire source doit avoir suffisamment d'or pour le transfert.
  - Le transfert échoue et la transaction est annulée si le solde est insuffisant ou si une autre transaction concurrente est en cours.

Pour tester la concurrence, vous pouvez simuler deux pirates lançant simultanément un POST /transferGold sur le même navire et observer que **les deux transactions échouent si elles se chevauchent**.

Livraison finale

- Le lien vers le **dépôt GitHub**.
- Une **courte vidéo (entre 10 et 15 minutes)** montrant
  - avec explications, ce que vous avez programmé dans le code (React Native et API)
  - les **tests Maestro**.
  - le lien de **déploiement fonctionnel (API + App mobile)**.
  - l'application en action.
  - avec explications le code et le fonctionnement en détail de votre transaction.

Remarques

- Faites de **petits commits fréquents**.
- Validez vos tests avant chaque *push*.
- Gardez votre code lisible et commenté.
- Gérez les erreurs d'API et les cas limites (connexion perdue, données manquantes, etc.).