Dart game

Enoncé

Vous avez eu la super idée d'acheter un jeu de fléchettes électroniques pour occuper les pauses dans votre boîte. Petit soucis, vous oubliez que Antonin (votre collègue un peu nul et bourrin) aurait de grandes chances de le casser en jouant avec. Quelques jours plus tard, ce qui devait arrivait arriva, et l'écran du jeu ne fonctionne plus. Vous êtes maintenant obligé de tenir les scores à la main.

C'est long et pénible, sans compter les erreurs de calculs. Heureusement vous êtes développeur, vous allez pouvoir faire un progamme qui gère les scores pour 3 modes de jeux.

Le TP se divise en 2 parties :

- **1ère Partie (Moteur)**: Gérer le calcul du score avec un programme en CLI (Affichage et input du score via le terminal), pour 3 modes de jeu différents. À rendre sur branche `engine` du git.
- **2ème Partie (API)**: Gérer les inputs via une API REST (Potentiellement utilisable par un ESP/Arduino branché au jeu de flechettes)

Tips:

- Lors de la partie 1, vous pouvez créez votre game engine dans un dossier `engine` que vous pourrez réutiliser sur la partie node. À la racine vous aurez un `cli.js` qui utilise inquirer et qui gère les appels à l'engine

1ÈRE PARTIE - MOTEUR

- Vous devez pouvoir choisir entre 2 et N joueurs, ainsi que le mode de jeu, au début de la partie
- Le joueur qui commence est choisi aléatoirement
- La partie doit pouvoir continuer jusqu'à définir l'ordre de l'ensemble des joueurs
- (Bonus) L'ordinateur annonce le nom de chaque joueur au début de son tour

Contraites techniques:

- Le programme doit être écrit en NodeJS
- Accessible entièrement depuis le terminal (astuce: vous pouvez utiliser console.table)
- L'architecture du moteur doit obligatoirement comporter des classes ainsi que des classes avec héritage (A vous de choisir l'architecture qui vous semble être la plus adaptée)
- Pour gérer l'input via le terminal, vous pouvez utiliser Inquirer
- Vous pouvez gérer les promesses comme vous le souhaitez

Règles du jeu à implémenter

- Le moteur devra gérer trois modes de jeu distincts avec un modèle de données qui leurs sont propres
- La cible est composée de 21 secteur distincts (numérotés de 1 à 20 plus le centre de la cible). Chaque secteur (hormis le centre de la cible) est composé de 3 multiplicateurs (x1, x2 et x3). Chaque secteur vaut autant de points que son numéro hormis le centre de la cible qui vaut 25 points. Il possède uniquement un multiplicateur x2. Le moteur devra donc gérer les différents secteurs ainsi que leurs multiplicateurs.
- Les trois modes de jeu à gérer sont le 301, le tour du monde et le cricket :
    - **N*1 - le tour du Monde**: Les joueurs lancent à tour de role trois flechettes chacun, en visant des secteurs spécifiques en commencant par 1, et en passant au suivant une fois que vous l'avez touché. Chaque partie du secteur compte (y compris les doubles et les triples). Chaque joueur doit frapper son secteur dans l'ordre croissant. Vous ne pouvez donc pas passer au secteur suivant tant que vous n'avez pas atteint celui que vous visez actuellement. La première personne à "faire le tour du monde" (atteindre tous les numéros, de 1 à 20) est la gagnante. Les multiplicateurs ne sont bien évidemment pas pris en compte.
    - **N*2 - le 301**: Chaque joueur commence avec un score initial de 301. Chacun leur tour, les joueur vont lancer trois fléchettes afin de baisser leur score. A chaque fléchette, on soustrait le score obtenu par la fléchettes du joueur à son score. Les multiplicateurs sont pris en compte dans le décompte du score. Le premier joueur qui arrive exactement à zéro a gagné. Si un joueur réalise plus de points qu'il n'en reste à soustraire, son tour n'est pas pris en compte. Attention, il faut absolument atteindre zéro en terminant par un double (On ne peut donc pas finir son tour avec un score égal à 1)
    - **N*3 - le Cricket**: Le jeu se joue uniquement en visant les chiffres 15 16 17 18 19 20 et la bulle. Le but du jeu est de « fermer » tous ces chiffres ainsi que la bulle en lançant trois fois la flèche dans ceux ci. Le gagnant est le premier joueur à avoir fermé tous les chiffres visés, en ayant au moins autant de points que les adversaires. Lorsqu'un joueur à atteint trois fois un chiffre, celui ci est fermé. La zone des doubles compte pour deux tirs, et la zone des triples pour trois tirs. Pour fermer la zone du 20 le joueur peut par exemple tirer trois fléchettes dans la zone simple, ou une dans le triple 20. Chaque joueur tire à tour de rôle une série de 3 fléchettes, en visant une des zones précitées. Si une fléchette atteint une autre zone, aucun point n'est marqué. S'il ferme une zone, et qu'un ou plusieurs autres joueurs ne l'ont pas encore fermé, il peut continuer à la viser. Chaque tir dans une zone déjà fermée lui rapportera alors autant de points que la valeur de la zone. Les adversaires par contre ne peuvent plus marquer de points grâce à cette zone. Elle appartient au joueur l'ayant fermé. Si tous les joueurs ont fermés une zone, plus personne ne peut y marquer de point.

2ÈME PARTIE - API Consigne

- doit être réalisée avec expressJS
- une BDD SQLite ou mongoDB doit être utilisée (à défaut, je vous conseille de commencer sans, avec une simple variable qui contiendra tous, cf. la variable `db` ici :https://gist.github.com/Tronix117/60ee47811c6e7f06c4224c529b9ed12a)
- La nomenclature ci-dessous doit être respectée.

- Le projet peut-être écrit en JS ou en TS (bonus pour TS si `strict: true`)
- Séparer les différentes couches logiques dans votre structure (propreté)
- Bonus si des appels sont traités en AJAX
- Pour tester les parties JSON, utilisez Postman

Note : `` ` correspond à l'objet `` { id: x, name: xxx, ... } `

API Ressources

Game

```
{
  id: number | string,
  mode: 'around-the-world' | '301' | 'cricket',
  name: string,
  currentPlayerId: null | string | number,
  status: 'draft' | 'started' | 'ended',
  createdAt: datetime,
}
```

Player

```
{
  id: number | string,
  name: string,
  email: string, // Format email à valider
  gameWin: number,
  gameLost: number,
  createdAt: datetime,
}
```

GamePlayer

```
{
  id?: number | string, // Optionnel
  playerId: number | string, // Le player doit exister
  gameId: number | string, // La game doit exister
  remainingShots: number | null, // Nombre de coup restant sur le tour de jeu
  score: number,
  rank: null | number, // La position de l'utilisateur à la fin de la partie
  order: number | null, // NULL par défaut, mais un ordre aléatoire est
assigné au démarrage de partie, commence à 0
  createdAt: datetime,
}
```

GameShot

```
{
  id: number | string,
  gameId: number | string,
  playerId: number | string,
  multiplicator: number, // 1, 2, 3
  sector: number, // BullEye = 25, en dehors = 0
  createdAt: datetime,
}
```

Le Serveur

La structure

```
assets/               <-- Les ressources statiques
  images/
    logo.png
  styles/
    main.css
  (scripts/)
    ...
  favicon.ico
routers/
  game.js
  (game/)             <-- Pour les sous-ressources
    (player.js)
    ...
  ...
models/               <-- Gère les modèles de donnée (échanges avec la base)
  Game.js
  Player.js
  ...
engine/
  gamemodes/
    around-the-world.js  <-- hérite de la classe abstraite
    cricket.js           <-- hérite de la classe abstraite
    301.js               <-- hérite de la classe abstraite
  gamemode.js         <-- Classe abstraite pour les 3 gamemodes
  (engine.js)         <-- au besoin
(errors/)
  (NotFound.js)
  ...
views/                <-- Les vues pour la partie HTML
  layout.XXX          <-- le layout (moteur de template libre, XXX => pug,
ejs, ...)
  ...
.gitignore
package.json
```

```
(db.js)               <-- connexion à la base
(config.js)           <-- si vous avez de la config
app.js                <-- Le point d'entrée du serveur Web
(cli.js)              <-- Si votre engine API et CLI sont coLe point d'entrée
de l'applicatif CLI (qui utilise)
router.js             <-- ce qui gère vos routes/initialize les routeur
(Dockerfile)
(docker-compose.yml)
```

Les fichiers/dossiers entre parenthèses sont facultatifs Les fichiers JS sont interchangeable avec des fichier TS

Gestion d'erreur

Lors d'erreurs, voici le format attendu :

En Json :

```
{
  error: {
    type: string, // Un code erreur unique (`CAPS_CAMEL_CASE`) selon le type
d'erreur, il peut être fournit dans la consigne
    message: string, // Un message à destination de l'utilisateur
  }
}
```

En HTML : un page Web, ce que vous voulez, qui affiche le message

Attention Certaines erreurs génériques ne sont pas indiquées dans la consigne, mais sont à gérer :

(Dans la liste, le chiffre est le code de status, le text est le type de l'erreur)

- `400 INVALID_FORMAT` Lorsque des données en entrées sont invalides
- `400 INVALID_FORMAT` Lorsque des données en entrées sont invalides
- `404 NOT_FOUND` Lorsque la ressource demandée n'existe pas
- `406 NOT_ACCEPTABLE` Le format attendu (heade Accept) n'est pas pris en charge
- `500 SERVEUR_ERROR` Erreur inconnue ou crash côté serveur

Liste des routes

Cf. détail section suivante

- GET /
- GET /players
- POST /players
- GET /players/new
- GET /players/{id}
- GET /players/{id}/edit
- PATCH /players/{id}

- DELETE /players/{id}
- GET /games
- GET /games/new
- POST /games
- GET /games/{id}
- GET /games/{id}/edit
- PATCH /games/{id}
- DELETE /games/{id}
- GET /games/{id}/players
- POST /games/{id}/players
- DELETE /games/{id}/players
- POST /games/{id}/shots
- DELETE /games/{id}/shots/previous
- GET /*

TIPS: Le routing

Exemple de routing.

Dans votre fichier `routes.js`, vous pouvez avoir :

```
const router = require('express').Router()
const gameRouter = require('./routers/game.js)

router.use('/games', gameRouter);

module.exports = router;
```

Il suffira simplement de `app.use` ce router dans votre fichier `app.js`

Et dans `routers/game.js`, vous pouvez avoir :

```
const router = require('express').Router()

router.get('/', (req, res, next) => {
  console.log('Correspond à /games');
})

module.exports = router
```

L'avantage de cette facon de faire, et que vous séparez bien chaque couche du serveur, de plus si demain, vous décidez de "franchiser" les routes, il suffira simplement de faire un `router.use('/parties', gameRouter)` pour que toutes les routes `/games` soient aussi accessibles depuis `/parties`

Détail des routes

Il est indispensable de bien respecter la nomenclature indiquée, les tests seront automatisés.

GET /

Réponse

En JSON Erreur `406 NOT_API_AVAILABLE`

En HTML: Redirige vers /games Attention à bien choisir le bon code de redirection

---

GET /players

Affiche la liste des joueurs

Requête

Paramètres query :

- `limit=number` : limite à un certain nombre de résultat (10 par défaut, 20 max)
- `page=number` : la page de résultat, commence à 1
- `sort=name`, `sort=email`, `sort=gameWin`, `sort=gameLost` : pour trier par un champs
- `reverse` : si le paramètre est présent, trie par ordre décroisant

Réponse

En JSON :

```
[
  <Player>,
  <Player>,
  ...
]
```

En HTML : Un tableau qui liste les joueurs avec un bouton pour accéder / supprimer le joueur

---

POST /players

Création d'un nouveau joueur

Requête

(JSON ou urlencoded)

```
{
  name: string,
  email: string,
}
```

Réponse

En JSON :

Code: `201`

```
<Player>
```

En HTML : redirection vers `/players/{id}`

GET /players/new

Affiche un formulaire de création de player (même vue que pour l'édition)

Réponse

En JSON : erreur `406 NOT_API_AVAILABLE` En HTML : le formulaire

GET /players/{id}

Réponse

En JSON:

```
<Player>
```

En HTML: redirection vers `GET /players/{id}/edit`

GET /players/{id}/edit

Affiche un formulaire de création de player (même vue que pour la création)

Réponse

En JSON : erreur `406 NOT_API_AVAILABLE` En HTML : le formulaire

PATCH /players/{id}

Permet d'éditer un utilisateur

Requête

(JSON ou urlencoded)

```
{
  name?: string,
  email?: string,
}
```

Le `?` définit que la propriété est optionnelle

Réponse

En JSON :

Code: `200`

```
<Player>
```

En HTML : redirection vers `/players`

DELETE /players/{id}

Permet de supprimer un joueur s'il n'est dans aucune partie dont le statut est 'started' ou 'ended'

Réponse

Erreurs possibles (autres que génériques) :

- `410 PLAYER_NOT_DELETABLE`, s'il n'est plus possible de supprimerle joueur

En JSON : code `204`, pas de body En HTML : redirection vers `/players`

GET /games

Affiche la liste des parties

Requête

Paramètres query :

- `limit=number` : limite à un certain nombre de résultat (10 par défaut, 20 max)
- `page=number` : la page de résultat, commence à 1
- `sort=name`, `sort=status` : pour trier par un champs
- `reverse` : si le paramètre est présent, trie par ordre décroisant
- `f.status=xxx` : pour filtrer par un état

Exemple :

```
GET /games?limit=3&page=1&sort=name&reverse&f.status=draft
```

Affichera les 3 premières parties non démarrées, par ordre alphabétique inversé

Réponse

En JSON :

```
[
  <Game>,
  <Game>,
  ...
]
```

En HTML :

- Un tableau qui liste les parties avec un bouton pour accéder / supprimer la partie
- Un bouton/lien "Gérer les joueurs" qui rediriges vers `GET /players`

GET /games/new

Note: ce type de route n'est pas RESTFull, mais RESTLike, acceptable, mais qui ne suit pas la convention REST

Affiche le formulaire de création de partie.

Réponse

En JSON : erreur `406 NOT_API_AVAILABLE` En HTML : le formulaire

POST /games

Création d'une nouvelle game

Requête

(JSON ou urlencoded)

```
{
  name: string,
  mode: 'around-the-world' | '301' | 'cricket',
}
```

Réponse

En JSON :

Code: `201`

```
<Game>
```

En HTML : redirection vers `/games/{id}`

GET /games/{id}

Réponse

En JSON :

```
{
  id: number | string,
  mode: 'around-the-world' | '301' | 'cricket',
  name: string,
  currentPlayerId: null | string | number,
  status: 'draft' | 'started' | 'ended',
  createdAt: datetime,
  enginePayload: { ... } // Données facultatives de l'engine, principalement
pour le cricket
}
```

En JSON **Si querystring `?include=gamePlayers`** :

```
{
  id: number | string,
  mode: 'around-the-world' | '301' | 'cricket',
  name: string,
  currentPlayerId: null | string | number,
  status: 'draft' | 'started' | 'ended',
  createdAt: datetime,
  enginePayload: { ... } // Données facultatives de l'engine, principalement
pour le cricket
  gamePlayers: [
    <GamePlayers>,
    <GamePlayers>,
    ...
  ],
}
```

En HTML :

- Affichage du state de la cible (brouillon, en cours, fini)
- Lien vers `GET /games/{id}/players` pour gérer les joueurs (si game est en draft)
- Affichage du tableau des joueurs
- Affichage du joueur actuel et de son nombre de coup restant pour le tour de jeu
- Affichage d'une cible, ou d'un tableau (multiplicateur / secteurs), chaque case pointe vers `POST /games/{id}/shots`

- Affichage des derniers coups joués (Bonus: bouton "Annuler le dernier tir")
- Bouton "Cible manquée"

Attention, selon le mode de jeu, la vue ou sous-vue (partial) utilisée pour l'affichage est susceptible d'être différente

---

GET /games/{id}/edit

Note: ce type de route n'est pas RESTFull, mais RESTLike, acceptable, mais qui ne suit pas la convention REST

Affiche le formulaire d'édition de partie. (cela peut être la même vue)

Réponse

En JSON : erreur `406 NOT_API_AVAILABLE` En HTML : le formulaire

PATCH /games/{id}

Permet d'éditer le nom et mode de jeu d'une game **non démarrée**

Permet également de lancer une game

Requête

(JSON ou urlencoded)

```
{
  name?: string,
  mode?: 'around-the-world' | '301' | 'cricket',
  status?: 'started',
}
```

Le `?` définit que la propriété est optionnelle

Réponse

Erreurs possibles (autre que génériques) :

- `410 GAME_NOT_EDITABLE`, s'il n'est plus possible d'éditer la game
- `422 GAME_NOT_STARTABLE`, si on essaie de lancer la game, et qu'elle est déjà lancée ou terminée
- `422 GAME_PLAYER_MISSING`, si on essaie de lancer la game et qu'il manque des joueurs

En JSON :

Code: `200`

```
<Game>
```

En HTML : redirection vers `/game/{id}`

DELETE /games/{id}

Permet de supprimer une partie.

Réponse

En JSON: code 204, pas de body

En HTML: redirection vers `GET /games`

GET /games/{id}/players

Affiche la liste des joueurs au sein d'une partie

Réponse

En JSON :

```
[
  <Player>,
  <Player>,
  ...
]
```

En HTML :

- Tableau Joueurs disponibles, avec bouton "Ajouter à la partie"
- Tableau Joueurs de la partie, avec bouton "Supprimer de la partie"

Tip: il n'est donc pas nécessaire de charger les joueurs disponible si on demande du JSON

POST /games/{id}/players

Permet d'ajouter un ou plusieurs joueurs à une partie non démarrée (en draft).

Requête

(JSON ou urlencoded)

```
[ playerId, playerId, ... ]
```

Ex: `[345, 432]`

Bonus: Créer des joueurs inexistants jusqu'à présent, ex: `[{ name: 'toto', email: 'toto@test.com'}, 345, 432]`

Réponse

Erreurs :

- `422 PLAYERS_NOT_ADDABLE_GAME_STARTED`, si la game a déjà commencé

En JSON: code `204`, pas de body

En HTML: redirection vers `GET /games/{id}/players`

DELETE /games/{id}/players

Permet de supprimer un ou plusieurs joueurs d'une partie non démarrée.

Requête

(JSON ou urlencoded)

Un body n'est pas recommandé sur un DELETE, par conséquent nous utiliserons la querystring :

Exemple: `DELETE /games/12/players?id=345&id=432`

Réponse

Erreurs :

- `422 PLAYERS_NOT_REMOVABLE_GAME_STARTED`, si la game a déjà commencé

En JSON: code 204, pas de body

En HTML: redirection vers `GET /games/{id}/players`

POST /games/{id}/shots

Requête

```
{
  sector: number,
  multiplicator: number,
}
```

Réponse

Erreurs:

- `422 GAME_NOT_STARTED`
- `422 GAME_ENDED`

En JSON : code 204, pas de body En HTML : redirection vers `GET /games/{id}`

DELETE /games/{id}/shots/previous

!!! ROUTE BONUS FACULTATIVE !!!

Annule le dernier coup joué (attention, si le joueur a changé entre temps, on doit revenir au jour précédent)

Réponse

En JSON : code 204, pas de body En HTML : redirection vers `GET /games/{id}`

GET /*

Permet d'envoyer les fichiers statiques qui sont dans le dossiers assets.

Ex:

- `GET /favicon.ico` (requis) -> retourne le fichier `assets/favicon.ico`
- `GET /styles/main.css` (requis) -> retourne le fichier `assets/styles/main.css`
- `GET /images/logo.png` (requis) -> retourne le fichier `assets/images/logo.png`
- `GET /scripts/*.js` (optionnel)
- `GET /...` -> tout autre fichier ou image statique