# === ENCART DE TRANSITION - ARCHITECTURE TECHNIQUE BRAINBATTLE - AGENT RESPONSABLE:INTERDICTION DE SUPPRIMER !!! ===

# AGENT RESPONSABLE: Architecte Technique - Dernière mise à jour: 16 mars 2025

#

# État Actuel & Contexte

# - Avancement: Document validé - Architecture technique finalisée avec priorisation mise à jour

# - Dernières actions réalisées:

# \\* Intégration des use cases dans l\'architecture

# \\* Mise à jour des modèles de données pour refléter tous les rôles utilisateurs

# \\* Ajout de l\'API pour l\'import initial de questions

# \\* Renforcement des points d\'extension pour l\'intégration IA future

# \\* Décision formelle sur la priorisation des fonctionnalités de jeu avant API réelle

# - Décisions techniques confirmées:

# \\* MERN Stack (MongoDB, Express, React, Node.js)

# \\* Socket.IO pour les communications temps réel

# \\* Tailwind CSS sans framework UI complet

# \\* Atomic Design pour la structure des composants

# \\* JWT pour l\'authentification avec rotation des tokens

# \\* Priorisation du développement des composants de jeu avant la préparation API réelle

#

# Dépendances & Relations

# - Ce document influence: Structure du projet, API endpoints, modèles de données, TaskSlicer

# - Ce document dépend de: Use cases, design system, tableau de bord (décisions)

# - Agents concernés: Développeur Frontend, Développeur Backend, TaskSlicer

#

# Attentes pour cet Agent

# 1. Fournir une architecture technique claire et complète pour guider l\'implémentation

# 2. Prévoir les points d\'extension pour le développement futur (notamment l\'IA)

# 3. Assurer la cohérence entre architecture frontend/backend et les use cases

# 4. Définir une structure de données adaptée aux différents types de questions

# 5. Proposer des solutions techniques optimisées tout en respectant le budget

# 6. Poser systématiquement des questions ou demander des documents supplémentaires pour toute mise à jour du document de référence

#

# Prochaines Étapes Identifiées

# 1. Collaborer avec le TaskSlicer pour assurer l\'alignement entre architecture et plan de développement

# 2. Commencer la recherche d\'une base de questions open-source adéquate

# 3. Préparer des recommandations techniques pour l\'implémentation de l\'API d\'import

# 4. Explorer les options d\'hébergement les plus économiques pour le MVP

# 5. Maintenir l'alignement avec le développeur frontend sur l'implémentation des composants de jeu

#

# Notes spécifiques

# - L\'architecture est désormais complète et reflète tous les use cases documentés

# - La gestion du thème \"Autre\" et l\'import de questions ont été spécifiquement traités

# - L\'architecture est modulaire pour permettre des évolutions futures sans refactoring majeur

# - DÉCISION DE PRIORISATION: Finaliser les composants de jeu (TS-35,36,37) avant la préparation API (TS-39)

# === FIN DE L'ENCART - AGENT RESPONSABLE:INTERDICTION DE SUPPRIMER !!! ===\*\* Architecture Technique - Quiz Entre Amis (BrainBattle)\*\*

## \*\*1. ARCHITECTURE GÉNÉRALE\*\*

### \*\*Architecture React/Node.js/MongoDB\*\*

Le choix d\'architecture MERN (MongoDB, Express, React, Node.js) est

excellent pour un projet de quiz en temps réel :

- \*\*Frontend (React)\*\* : Parfait pour une interface interactive et

> réactive

- \*\*Backend (Node.js)\*\* : Idéal pour les applications temps réel grâce

> à son modèle asynchrone

- \*\*Base de données (MongoDB)\*\* : Bien adapté pour stocker des données

> hétérogènes comme les différents types de questions

### \*\*Structure du projet\*\*

quiz-entre-amis/

├── client/ \# Frontend React

│ ├── public/

│ ├── src/

│ │ ├── assets/ \# Images, fonts, etc.

│ │ ├── components/ \# Composants React (Atomic Design)

│ │ │ ├── atoms/

│ │ │ ├── molecules/

│ │ │ ├── organisms/

│ │ │ ├── templates/

│ │ │ └── pages/

│ │ ├── context/ \# Context API pour la gestion d\'état

│ │ ├── hooks/ \# Custom hooks

│ │ ├── services/ \# Services API et WebSocket

│ │ ├── utils/ \# Fonctions utilitaires

│ │ ├── App.js

│ │ └── index.js

├── server/ \# Backend Node.js

│ ├── controllers/ \# Logique métier

│ ├── middleware/ \# Middleware (auth, validation, etc.)

│ ├── models/ \# Modèles MongoDB

│ ├── routes/ \# Routes API REST

│ ├── services/ \# Services métier

│ ├── socket/ \# Logique WebSocket

│ ├── config/ \# Configuration

│ ├── utils/ \# Fonctions utilitaires

│ ├── scripts/ \# Scripts d\'initialisation et utilitaires

│ ├── app.js \# Configuration Express

│ └── server.js \# Point d\'entrée

└── package.json

### \*\*Atomic Design pour React\*\*

L\'approche Atomic Design est parfaite pour un projet évolutif comme

BrainBattle. Cette méthodologie est particulièrement pertinente pour

gérer les nombreux composants UI identifiés dans le design system :

- \*\*Atoms\*\* : Boutons, inputs, labels, icônes, badges, timer,

> indicateurs de difficulté

- \*\*Molecules\*\* : Cartes de question, cartes de thème, formulaires

> d\'authentification, barres de progression, contrôles audio/vidéo

- \*\*Organisms\*\* : Liste de questions, tableaux de score, modal de fin

> de partie, interface de création/validation de questions

- \*\*Templates\*\* : Layout de quiz, écran de lobby, page de création de

> quiz, interfaces admin et validateur

- \*\*Pages\*\* : Correspond aux différentes pages identifiées dans les

> use cases (accueil non connecté, accueil connecté, inscription,

> connexion, etc.)

## \*\*2. CHOIX TECHNIQUES SPÉCIFIQUES\*\*

### \*\*Communication temps réel\*\*

Socket.IO est recommandé pour la communication en temps réel,

essentielle pour les fonctionnalités de jeu multi-joueurs :

- \*\*Avantages\*\* :

- Fallback automatique en cas d\'échec des WebSockets

- API simple et bien documentée

- Support des salles (rooms) idéal pour les sessions de quiz

- Bibliothèque mature et stable

// Côté serveur (Node.js)

const io = require(\'socket.io\')(server, {

cors: {

origin: process.env.CLIENT\_URL,

methods: \[\"GET\", \"POST\"\]

}

});

io.on(\'connection\', (socket) =\> {

// Rejoindre une salle (partie de quiz)

socket.on(\'joinGame\', (gameId) =\> {

socket.join(gameId);

io.to(gameId).emit(\'playerJoined\', { playerId: socket.id });

});

// Soumettre une réponse

socket.on(\'submitAnswer\', (data) =\> {

// Traitement de la réponse

io.to(data.gameId).emit(\'answerSubmitted\', {

playerId: socket.id,

correct: result.correct,

score: result.score

});

});

});

// Côté client (React)

import { io } from \"socket.io-client\";

import { useEffect, useState } from \"react\";

const GameRoom = ({ gameId }) =\> {

const \[socket, setSocket\] = useState(null);

useEffect(() =\> {

const newSocket = io(process.env.REACT\_APP\_API\_URL);

setSocket(newSocket);

// Rejoindre la partie

newSocket.emit(\'joinGame\', gameId);

// Écouter les événements

newSocket.on(\'playerJoined\', (data) =\> {

// Mettre à jour l\'état local

});

return () =\> newSocket.disconnect();

}, \[gameId\]);

const submitAnswer = (answerId) =\> {

socket.emit(\'submitAnswer\', { gameId, answerId });

};

// \...

};

### \*\*Recommandations pour faciliter l\'évolution future\*\*

1. \*\*Architecture orientée microservices\*\*

Bien que le MVP puisse commencer comme une application monolithique,

structurez-le en modules distincts qui pourront être extraits en

microservices ultérieurement :

server/

├── core/ \# Noyau commun

├── modules/

│ ├── auth/ \# Module d\'authentification

│ ├── questions/ \# Module de gestion des questions

│ ├── games/ \# Module de gestion des parties

│ └── ai/ \# Module IA (vide pour le MVP, préparé pour le futur)

2. \*\*Configuration par environnement\*\*

Préparez des configurations distinctes pour activer/désactiver les

fonctionnalités IA :

// server/config/features.js

module.exports = {

aiGeneration: process.env.ENABLE\_AI\_GENERATION === \'true\',

aiValidation: process.env.ENABLE\_AI\_VALIDATION === \'true\',

aiDifficulty: process.env.ENABLE\_AI\_DIFFICULTY === \'true\'

};

3. \*\*Endpoints d\'extension IA (désactivés dans le MVP)\*\*

// server/routes/question.routes.js

const express = require(\'express\');

const router = express.Router();

const questionController =

require(\'../controllers/question.controller\');

const auth = require(\'../middleware/auth\');

const features = require(\'../config/features\');

// Routes standard

router.get(\'/\', questionController.getQuestions);

router.get(\'/:id\', questionController.getQuestion);

router.post(\'/\', auth, questionController.createQuestion);

router.put(\'/:id\', auth, questionController.updateQuestion);

router.delete(\'/:id\', auth, questionController.deleteQuestion);

// Routes futures IA (désactivées par défaut)

if (features.aiGeneration) {

router.post(\'/generate\', auth, questionController.generateQuestion);

router.post(\'/improve/:id\', auth, questionController.improveQuestion);

}

module.exports = router;

4. \*\*Interface frontend préparée pour l\'IA\*\*

Intégrez dans le frontend des éléments d\'interface UI désactivés mais

préparés pour l\'IA :

// client/src/components/organisms/QuestionCreator.jsx

import React from \'react\';

import { useFeatures } from \'../../hooks/useFeatures\';

import { Button, FormControl, Input, Textarea } from \'../atoms\';

import { QuestionForm } from \'../molecules\';

const QuestionCreator = () =\> {

const { aiGeneration } = useFeatures();

return (

\<div className=\"question-creator\"\>

\<h2\>Créer une question\</h2\>

{/\\* Formulaire standard \\*/}

\<QuestionForm /\>

{/\\* Interface IA désactivée dans le MVP \\*/}

{aiGeneration && (

\<div className=\"ai-generator\"\>

\<h3\>Ou générer avec IA\</h3\>

\<FormControl\>

\<label\>Prompt pour l\'IA\</label\>

\<Textarea placeholder=\"Décrivez la question que vous souhaitez

générer\...\" /\>

\</FormControl\>

\<Button variant=\"primary\"\>Générer avec IA\</Button\>

\</div\>

)}

\</div\>

);

};

5. \*\*Hooks prêts pour l\'IA\*\*

// client/src/hooks/useFeatures.js

import { useState, useEffect } from \'react\';

import api from \'../services/api\';

export const useFeatures = () =\> {

const \[features, setFeatures\] = useState({

aiGeneration: false,

aiValidation: false,

aiDifficulty: false

});

useEffect(() =\> {

const fetchFeatures = async () =\> {

try {

const response = await api.get(\'/api/config/features\');

setFeatures(response.data);

} catch (error) {

console.error(\'Failed to fetch features:\', error);

}

};

fetchFeatures();

}, \[\]);

return features;

};

## \*\*8. PRIORISATION DU DÉVELOPPEMENT\*\*

### \*\*Séquence de développement recommandée\*\*

Suite à la décision stratégique prise le 16 mars 2025, il est recommandé de prioriser le développement dans l'ordre suivant:

1. \*\*Phase 1 (Priorité Haute)\*\*: Finaliser le développement des composants et fonctionnalités de jeu

- Développer tous les composants de jeu essentiels (TS-35)

- Implémenter la page de jeu complète (TS-36)

- Créer la salle d'attente (TS-37)

- Simuler les interactions temps réel avec des données factices

2. \*\*Phase 2 (Priorité Moyenne)\*\*: Préparer l'intégration avec l'API réelle (TS-39)

- Créer les interfaces pour tous les services

- Développer le pattern d'adaptateur comme décrit dans l'architecture

- Mettre en place la factory de services

- Documenter clairement tous les endpoints d'API

3. \*\*Phase 3 (Priorité Basse)\*\*: Fonctionnalités additionnelles

- Page de statistiques utilisateur (TS-40)

- Améliorations d'accessibilité et d'UX (TS-41)

- Optimisations de performance

### \*\*Justification de cette priorisation\*\*

La décision de compléter d'abord les fonctionnalités de jeu avant la préparation de l'API réelle se base sur les facteurs suivants:

1. \*\*Cohérence de l'expérience utilisateur\*\*: Finir l'ensemble des fonctionnalités de jeu permettra de tester l'expérience utilisateur complète, même avec des données factices.

2. \*\*Identification complète des besoins d'API\*\*: En développant toutes les interfaces utilisateur et fonctionnalités en premier, nous aurons une vision précise et exhaustive des besoins réels en matière d'API.

3. \*\*Architecture déjà préparée pour l'API\*\*: Notre architecture technique a déjà prévu les points d'extension nécessaires pour une intégration API future, ce qui minimise les risques de redesign majeur.

4. \*\*Efficacité du développement\*\*: Cette approche permet au développeur frontend de rester concentré sur un ensemble cohérent de tâches sans basculer entre différents domaines techniques.

5. \*\*Tests plus efficaces\*\*: Les tests pourront d'abord valider les fonctionnalités de l'interface utilisateur indépendamment, puis l'intégration avec l'API.

### \*\*Impact sur la planification\*\*

Cette décision confirme l'ordre des tâches déjà suggéré dans le tableau de bord et permet au TaskSlicer de poursuivre le découpage actuel sans changement majeur. Le Développeur Frontend peut continuer à se concentrer sur les tâches TS-35, TS-36 et TS-37 sans avoir à préparer immédiatement l'intégration API.

## \*\*6. SOLUTIONS CLOUD ÉCONOMIQUES\*\*

Pour respecter votre budget maximal de 100€, voici mes recommandations :

1. \*\*Hébergement\*\* :

- \*\*Render\*\* : Plan gratuit pour le développement, puis 7€/mois

> pour le backend

- \*\*Vercel\*\* : Plan gratuit pour le frontend React

- \*\*MongoDB Atlas\*\* : Plan gratuit (512MB) suffisant pour le MVP

2. \*\*Mise en cache\*\* :

- Utiliser \*\*Redis Cloud\*\* (plan gratuit jusqu\'à 30MB) pour

> optimiser les performances

3. \*\*Solutions d\'optimisation\*\* :

- Compression des assets avec \*\*gzip/Brotli\*\*

- Mise en cache côté navigateur

- Lazy loading des composants React

Total mensuel estimé : \~7-10€, bien en-dessous de votre budget de 100€.

## \*\*7. CONCLUSION\*\*

Le projet BrainBattle (Quiz Entre Amis) bénéficiera grandement de cette

architecture modulaire, qui facilite les itérations futures et permet

une évolution progressive. L\'architecture proposée prend en compte tous

les use cases identifiés et les contraintes techniques, notamment :

1. L\'expérience utilisateur fluide grâce à Socket.IO pour les

> communications en temps réel

2. La gestion d\'état optimisée avec Context API et Zustand

3. Un système d\'authentification robuste et sécurisé

4. Une API complète, incluant la validation et l\'import de questions

5. Un design system cohérent implémenté avec Tailwind CSS

6. Des points d\'extension permettant l\'ajout futur de fonctionnalités

> de génération IA

Pour les prochaines phases, nous recommandons de :

1. Commencer par un MVP ciblant uniquement le desktop (comme prévu)

2. Identifier et intégrer une source de questions open-source de

> qualité (tâche critique S2-08)

3. Utiliser une approche itérative avec tests fréquents

4. Prévoir l\'ajout de fonctionnalités sociales et mobiles dans les

> versions futures

Cette architecture est conçue pour permettre l\'ajout progressif de

fonctionnalités sans refactoring majeur, tout en maintenant de bonnes

performances même avec une augmentation du nombre d\'utilisateurs.

### \*\*Gestion d\'état React\*\*

Pour la gestion d\'état, une approche hybride est recommandée

conformément aux besoins identifiés dans les use cases :

1. \*\*Context API\*\* pour l\'état global (utilisateur connecté, thème,

> configuration)

2. \*\*Zustand\*\* pour les états complexes (déroulement du quiz, scores,

> progression)

// store.js (Zustand)

import create from \'zustand\';

export const useGameStore = create((set) =\> ({

gameState: \'waiting\',

players: \[\],

currentQuestion: null,

scores: {},

startGame: () =\> set({ gameState: \'playing\' }),

setCurrentQuestion: (question) =\> set({ currentQuestion: question }),

updateScore: (playerId, score) =\> set((state) =\> ({

scores: { \...state.scores, \[playerId\]: score }

})),

}));

// AuthContext.js (Context API)

import { createContext, useState, useContext, useEffect } from

\'react\';

const AuthContext = createContext();

export const AuthProvider = ({ children }) =\> {

const \[user, setUser\] = useState(null);

const \[loading, setLoading\] = useState(true);

// Vérifier le token au chargement

useEffect(() =\> {

const checkAuth = async () =\> {

const token = localStorage.getItem(\'token\');

if (token) {

try {

// Vérifier la validité du token

const response = await fetch(\'/api/auth/verify\', {

headers: { Authorization: \`Bearer \${token}\` }

});

if (response.ok) {

const userData = await response.json();

setUser(userData);

} else {

localStorage.removeItem(\'token\');

}

} catch (error) {

console.error(\'Auth error:\', error);

}

}

setLoading(false);

};

checkAuth();

}, \[\]);

const login = async (credentials) =\> {

// Logique de connexion

};

const logout = () =\> {

localStorage.removeItem(\'token\');

setUser(null);

};

return (

\<AuthContext.Provider value={{ user, login, logout, loading }}\>

{children}

\</AuthContext.Provider\>

);

};

export const useAuth = () =\> useContext(AuthContext);

### \*\*Authentification JWT\*\*

Stratégie d\'authentification adaptée aux exigences de sécurité et aux

use cases de connexion/inscription :

1. \*\*Stockage\*\* : LocalStorage pour le token d\'accès, HttpOnly cookie

> pour le refresh token

2. \*\*Rafraîchissement\*\* : Utiliser un token d\'accès avec durée courte

> (15-30 min) et un refresh token plus long (7 jours)

3. \*\*Sécurité\*\* : Rotation des refresh tokens, invalidation côté

> serveur

// Middleware d\'authentification (server/middleware/auth.js)

const jwt = require(\'jsonwebtoken\');

const auth = (req, res, next) =\> {

try {

const token = req.header(\'Authorization\').replace(\'Bearer \', \'\');

const decoded = jwt.verify(token, process.env.JWT\_SECRET);

req.userId = decoded.userId;

next();

} catch (error) {

res.status(401).json({ message: \'Authentification requise\' });

}

};

module.exports = auth;

// Service d\'authentification (server/services/auth.service.js)

const generateTokens = (userId) =\> {

const accessToken = jwt.sign(

{ userId },

process.env.JWT\_SECRET,

{ expiresIn: \'30m\' }

);

const refreshToken = jwt.sign(

{ userId },

process.env.JWT\_REFRESH\_SECRET,

{ expiresIn: \'7d\' }

);

return { accessToken, refreshToken };

};

const refreshAuth = async (refreshToken) =\> {

try {

const decoded = jwt.verify(refreshToken,

process.env.JWT\_REFRESH\_SECRET);

// Vérifier si le refresh token est valide en base

const user = await User.findById(decoded.userId);

if (!user \|\| !user.refreshTokens.includes(refreshToken)) {

throw new Error(\'Token invalide\');

}

// Générer nouveaux tokens

const tokens = generateTokens(user.\\_id);

// Rotation du refresh token

await User.findByIdAndUpdate(user.\\_id, {

\$pull: { refreshTokens: refreshToken },

\$push: { refreshTokens: tokens.refreshToken }

});

return tokens;

} catch (error) {

throw new Error(\'Rafraîchissement échoué\');

}

};

### \*\*UI Framework et Styling\*\*

Conformément à la décision prise lors du Sprint 1 (documentée dans le

tableau de bord) :

- Utilisation de \*\*Tailwind CSS\*\* sans framework UI complet

- Possibilité d\'intégrer des composants spécifiques de Headless

> UI/Radix UI si nécessaire

- Implémentation du design system documenté dans les fichiers HTML

> fournis

Cette approche permet de rester dans les contraintes budgétaires tout en

offrant une flexibilité suffisante pour implémenter les interfaces

complexes identifiées dans les use cases.

## \*\*3. STRUCTURE DES API\*\*

### \*\*Endpoints RESTful\*\*

#### \*\*Utilisateurs et authentification\*\*

POST /api/auth/register \# Inscription

POST /api/auth/login \# Connexion

POST /api/auth/refresh \# Rafraîchir token

POST /api/auth/logout \# Déconnexion

GET /api/users/me \# Profil utilisateur

PUT /api/users/me \# Mise à jour profil

#### \*\*Questions et thèmes\*\*

GET /api/themes \# Liste des thèmes

GET /api/themes/:id \# Détails d\'un thème

POST /api/themes \# Créer un thème

PUT /api/themes/:id \# Modifier un thème

DELETE /api/themes/:id \# Supprimer un thème

GET /api/questions \# Liste des questions (filtrable)

GET /api/questions/:id \# Détails d\'une question

POST /api/questions \# Créer une question

PUT /api/questions/:id \# Modifier une question

DELETE /api/questions/:id \# Supprimer une question

#### \*\*Gestion des parties\*\*

POST /api/games \# Créer une partie

GET /api/games/:id \# Obtenir les détails d\'une partie

POST /api/games/:id/join \# Rejoindre une partie

PUT /api/games/:id/start \# Démarrer une partie

PUT /api/games/:id/end \# Terminer une partie

GET /api/games/:id/results \# Résultats d\'une partie

#### \*\*Administration et validation\*\*

GET /api/admin/users \# Liste des utilisateurs (admin)

PUT /api/admin/users/:id \# Modifier un utilisateur (admin)

DELETE /api/admin/users/:id \# Supprimer un utilisateur (admin)

GET /api/validate/questions \# Questions en attente de validation

PUT /api/validate/questions/:id/approve \# Approuver une question

PUT /api/validate/questions/:id/reject \# Rejeter une question

### \*\*API pour l\'import initial de questions open-source\*\*

Pour l\'import initial de questions open-source (identifié comme une

tâche critique dans le tableau de bord) :

POST /api/import/questions

Format du payload JSON :

{

\"source\": \"open\_trivia\_db\", // source des questions

\"format\": \"json\", // format des données (json, csv)

\"questions\": \[

{

\"text\": \"Quelle est la capitale de la France?\",

\"type\": \"qcm\",

\"theme\": \"Géographie\", // Sera associé ou créé si nécessaire

\"difficulty\": \"easy\",

\"options\": \[

{ \"text\": \"Paris\", \"isCorrect\": true },

{ \"text\": \"Lyon\", \"isCorrect\": false },

{ \"text\": \"Marseille\", \"isCorrect\": false },

{ \"text\": \"Toulouse\", \"isCorrect\": false }

\],

\"timeLimit\": 20,

\"points\": 100

}

// \...autres questions

\]

}

Pour la validation des données importées :

// server/services/import.service.js

const validateQuestionImport = (question) =\> {

const errors = \[\];

// Validations obligatoires

if (!question.text) errors.push(\'Le texte de la question est requis\');

if (!question.type) errors.push(\'Le type de question est requis\');

// Validations spécifiques au type

switch (question.type) {

case \'qcm\':

if (!question.options \|\| !Array.isArray(question.options) \|\|

question.options.length \< 2) {

errors.push(\'Les questions QCM doivent avoir au moins 2 options\');

}

if (!question.options.some(option =\> option.isCorrect === true)) {

errors.push(\'Au moins une option doit être correcte\');

}

break;

case \'text\':

if (!question.correctAnswer) {

errors.push(\'La réponse correcte est requise pour les questions de type

texte\');

}

break;

case \'timeline\':

if (!question.date) {

errors.push(\'La date est requise pour les questions de type

timeline\');

}

break;

case \'drag-n-drop\':

if (!question.options \|\| !question.options.every(opt =\> opt.position

!== undefined)) {

errors.push(\'Les positions sont requises pour les options des questions

drag-n-drop\');

}

break;

}

return errors;

};

const processQuestionImport = async (questions, userId) =\> {

const results = {

total: questions.length,

imported: 0,

failed: 0,

errors: \[\]

};

for (const q of questions) {

const errors = validateQuestionImport(q);

if (errors.length === 0) {

try {

// Trouver ou créer le thème

let themeId;

if (q.theme) {

const theme = await Theme.findOne({ name: q.theme });

if (theme) {

themeId = theme.\\_id;

} else {

// Créer un nouveau thème

const newTheme = await Theme.create({

name: q.theme,

description: \`Thème \${q.theme}\`,

createdBy: userId

});

themeId = newTheme.\\_id;

}

} else {

// Utiliser le thème \"Autre\"

const otherTheme = await Theme.findOne({ name: \'Autre\' });

themeId = otherTheme.\\_id;

}

// Créer la question

await Question.create({

text: q.text,

type: q.type,

themeId,

options: q.options \|\| \[\],

correctAnswer: q.correctAnswer,

date: q.date,

difficulty: q.difficulty \|\| \'medium\',

timeLimit: q.timeLimit \|\| 20,

points: q.points \|\| 100,

createdBy: userId

});

results.imported++;

} catch (error) {

results.failed++;

results.errors.push(\`Erreur pour la question \"\${q.text}\":

\${error.message}\`);

}

} else {

results.failed++;

results.errors.push(\`Validation échouée pour la question

\"\${q.text}\": \${errors.join(\', \')}\`);

}

}

return results;

};

### \*\*Format des payloads WebSocket\*\*

Les communications WebSocket sont essentielles pour les fonctionnalités

temps réel identifiées dans les use cases, notamment les phases de jeu

et la visualisation de l\'état des autres joueurs :

// Événements WebSocket

// 1. Événements de gestion de partie

{

event: \'game:join\',

data: {

gameId: \'abc123\',

player: {

id: \'user123\',

name: \'Jean\'

}

}

}

// 2. Événements de jeu

{

event: \'game:question\',

data: {

questionId: \'q123\',

text: \'Quelle est la capitale de la France?\',

type: \'qcm\',

options: \[

{ id: \'a\', text: \'Paris\' },

{ id: \'b\', text: \'Lyon\' },

{ id: \'c\', text: \'Marseille\' }

\],

timeLimit: 20

}

}

// 3. Événements de réponse

{

event: \'game:answer\',

data: {

playerId: \'user123\',

questionId: \'q123\',

answer: \'a\',

timeSpent: 5.2

}

}

// 4. Événements de résultats

{

event: \'game:results\',

data: {

questionId: \'q123\',

correctAnswer: \'a\',

playerResults: \[

{ playerId: \'user123\', correct: true, timeSpent: 5.2, score: 150 },

{ playerId: \'user456\', correct: false, timeSpent: 8.7, score: 0 }

\]

}

}

## \*\*4. MODÈLES DE DONNÉES\*\*

### \*\*Schémas MongoDB\*\*

#### \*\*Utilisateurs\*\*

// server/models/user.model.js

const mongoose = require(\'mongoose\');

const bcrypt = require(\'bcrypt\');

const userSchema = new mongoose.Schema({

username: {

type: String,

required: true,

unique: true,

trim: true,

minlength: 3,

maxlength: 20

},

email: {

type: String,

required: true,

unique: true,

trim: true,

lowercase: true

},

password: {

type: String,

required: true,

minlength: 8

},

role: {

type: String,

enum: \[\'user\', \'validator\', \'admin\'\],

default: \'user\'

},

profilePicture: {

type: String,

default: null

},

createdQuizzes: \[{

type: mongoose.Schema.Types.ObjectId,

ref: \'Theme\'

}\],

gamesPlayed: {

type: Number,

default: 0

},

totalScore: {

type: Number,

default: 0

},

refreshTokens: \[{

type: String

}\],

isActive: {

type: Boolean,

default: true

},

createdAt: {

type: Date,

default: Date.now

}

});

// Hash password before saving

userSchema.pre(\'save\', async function(next) {

if (!this.isModified(\'password\')) return next();

this.password = await bcrypt.hash(this.password, 10);

next();

});

// Method to compare passwords

userSchema.methods.comparePassword = async function(candidatePassword) {

return bcrypt.compare(candidatePassword, this.password);

};

module.exports = mongoose.model(\'User\', userSchema);

#### \*\*Questions\*\*

// server/models/question.model.js

const mongoose = require(\'mongoose\');

const optionSchema = new mongoose.Schema({

text: {

type: String,

required: true

},

isCorrect: {

type: Boolean,

required: true

},

// Pour les questions de type drag-n-drop

position: {

type: Number

}

});

const questionSchema = new mongoose.Schema({

text: {

type: String,

required: true

},

type: {

type: String,

enum: \[\'qcm\', \'text\', \'timeline\', \'drag-n-drop\'\],

required: true

},

themeId: {

type: mongoose.Schema.Types.ObjectId,

ref: \'Theme\',

required: true

},

options: \[optionSchema\],

// Pour les questions de type texte

correctAnswer: {

type: String

},

alternativeAnswers: \[String\], // Réponses alternatives acceptées

// Pour les questions de type timeline

date: {

type: Date

},

difficulty: {

type: String,

enum: \[\'easy\', \'medium\', \'hard\'\],

default: \'medium\'

},

timeLimit: {

type: Number,

default: 20,

min: 5,

max: 60

},

points: {

type: Number,

default: 100

},

media: {

type: {

type: String,

enum: \[\'image\', \'audio\', null\],

default: null

},

url: String

},

createdBy: {

type: mongoose.Schema.Types.ObjectId,

ref: \'User\',

required: true

},

// Champs pour validation

status: {

type: String,

enum: \[\'pending\', \'approved\', \'rejected\'\],

default: \'pending\'

},

validatedBy: {

type: mongoose.Schema.Types.ObjectId,

ref: \'User\'

},

validationComment: String,

validatedAt: Date,

// Statistiques

timesUsed: {

type: Number,

default: 0

},

correctAnswerRate: {

type: Number,

default: 0

},

// Métadonnées pour les futures questions IA

aiGenerated: {

type: Boolean,

default: false

},

aiPrompt: String,

aiModel: String,

createdAt: {

type: Date,

default: Date.now

}

});

module.exports = mongoose.model(\'Question\', questionSchema);

#### \*\*Thèmes\*\*

// server/models/theme.model.js

const mongoose = require(\'mongoose\');

const themeSchema = new mongoose.Schema({

name: {

type: String,

required: true,

unique: true,

trim: true

},

description: {

type: String,

required: true

},

image: {

type: String

},

icon: {

type: String,

default: \'📚\' // Emoji par défaut

},

isActive: {

type: Boolean,

default: true

},

isPublic: {

type: Boolean,

default: true

},

// Attribut pour identifier le thème \"Autre\"

isDefault: {

type: Boolean,

default: false

},

createdBy: {

type: mongoose.Schema.Types.ObjectId,

ref: \'User\',

required: true

},

questionsCount: {

type: Number,

default: 0,

validate: {

validator: function(count) {

// Exception pour le thème \"Autre\" qui peut avoir moins de 20

questions

return this.isDefault \|\| count \>= 20 \|\| this.isNew;

},

message: \'Un thème doit contenir au moins 20 questions\'

}

},

timesPlayed: {

type: Number,

default: 0

},

avgRating: {

type: Number,

default: 0

},

createdAt: {

type: Date,

default: Date.now

}

});

// Middleware pour créer le thème \"Autre\" s\'il n\'existe pas

themeSchema.statics.ensureDefaultThemeExists = async

function(adminUserId) {

const defaultTheme = await this.findOne({ isDefault: true });

if (!defaultTheme) {

await this.create({

name: \'Autre\',

description: \'Questions sans thème spécifique\',

isPublic: true,

isDefault: true,

createdBy: adminUserId

});

}

};

module.exports = mongoose.model(\'Theme\', themeSchema);

Script d\'initialisation pour assurer l\'existence du thème \"Autre\" :

// server/scripts/init-db.js

const User = require(\'../models/user.model\');

const Theme = require(\'../models/theme.model\');

const initDatabase = async () =\> {

try {

// Vérifier si un utilisateur admin existe

let adminUser = await User.findOne({ role: \'admin\' });

// Si aucun admin n\'existe, en créer un par défaut

if (!adminUser) {

adminUser = await User.create({

username: \'admin\',

email: process.env.ADMIN\_EMAIL \|\| \'admin@quizentreamis.fr\',

password: process.env.ADMIN\_PASSWORD \|\| \'ChangeMe!2023\',

role: \'admin\'

});

console.log(\'Admin user created\');

}

// S\'assurer que le thème \"Autre\" existe

await Theme.ensureDefaultThemeExists(adminUser.\\_id);

console.log(\'Default theme checked\');

console.log(\'Database initialization completed\');

} catch (error) {

console.error(\'Database initialization failed:\', error);

}

};

module.exports = initDatabase;

#### \*\*Parties\*\*

// server/models/game.model.js

const mongoose = require(\'mongoose\');

const playerSchema = new mongoose.Schema({

userId: {

type: mongoose.Schema.Types.ObjectId,

ref: \'User\'

},

// Pour les joueurs invités sans compte

name: {

type: String

},

isHost: {

type: Boolean,

default: false

},

isReady: {

type: Boolean,

default: false

},

score: {

type: Number,

default: 0

},

answers: \[{

questionId: {

type: mongoose.Schema.Types.ObjectId,

ref: \'Question\'

},

answer: mongoose.Schema.Types.Mixed,

correct: Boolean,

timeSpent: Number,

points: Number

}\],

rank: {

type: Number

}

});

const gameSchema = new mongoose.Schema({

code: {

type: String,

required: true,

unique: true

},

themes: \[{

type: mongoose.Schema.Types.ObjectId,

ref: \'Theme\'

}\],

status: {

type: String,

enum: \[\'waiting\', \'playing\', \'correction\', \'finished\'\],

default: \'waiting\'

},

settings: {

questionsCount: {

type: Number,

default: 10,

min: 5,

max: 50

},

timePerQuestion: {

type: Number,

default: 20,

min: 5,

max: 60

},

randomOrder: {

type: Boolean,

default: true

},

showLeaderboardAfterEach: {

type: Boolean,

default: true

},

allowGuests: {

type: Boolean,

default: true

}

},

players: \[playerSchema\],

questions: \[{

questionId: {

type: mongoose.Schema.Types.ObjectId,

ref: \'Question\'

},

startedAt: Date,

endedAt: Date

}\],

currentQuestionIndex: {

type: Number,

default: -1

},

createdBy: {

type: mongoose.Schema.Types.ObjectId,

ref: \'User\',

required: true

},

createdAt: {

type: Date,

default: Date.now

},

startedAt: {

type: Date

},

endedAt: {

type: Date

}

});

// Générer un code unique pour la partie

gameSchema.pre(\'save\', async function(next) {

if (this.isNew) {

// Générer un code alphanumérique de 6 caractères

const characters = \'ABCDEFGHJKLMNPQRSTUVWXYZ23456789\';

let code = \'\';

for (let i = 0; i \< 6; i++) {

code += characters.charAt(Math.floor(Math.random() \\*

characters.length));

}

this.code = code;

}

next();

});

module.exports = mongoose.model(\'Game\', gameSchema);

## \*\*5. PRÉPARATION À L\'ÉVOLUTION FUTURE\*\*

### \*\*Structure pour l\'ajout ultérieur d\'un système de génération IA\*\*

Pour faciliter l\'intégration future d\'un système de génération IA

(comme identifié dans les contraintes du projet), je recommande de

mettre en place une architecture à base de services avec des interfaces

bien définies :

graph TD

A\[Client React\] \--\> B\[API Gateway\]

B \--\> C\[Service Questions\]

B \--\> D\[Service Utilisateurs\]

B \--\> E\[Service Parties\]

B \--\> F\[WebSocket Manager\]

C \--\> G\[(Database MongoDB)\]

D \--\> G

E \--\> G

C \--\> H\[Question Factory\]

H \--\> I\[Manual Question Creator\]

H \--\> J\[AI Question Generator\]

I \--\> K\[Question Validator\]

J \--\> K

K \--\> C

subgraph \"Future IA Integration\"

J

L\[OpenAI API\]

M\[Custom ML Model\]

J \--\> L

J \--\> M

end