Vannes
iut
Université Bretagne Sud

Cahier des Tests du projet MathsALaMaison (Application et Serveur)

TABLE DES MATIERES

INTRODUCTION

Ce présent document contient les différents tests réalisées dans le cadre du projet MathsALaMaison.

Les tests sont réparties en trois catégories principales :

1. Tests réalisées sur l'application React-Native : Ces tests sont les tests effectuées après le développement de chaque nouvelle fonctionnalités.
2. Tests réalisées sur le serveur : Ces tests concerne les tests effectuées sur le serveur de jeu, l'API, l'intégration de l'application et la base de données.
3. Test en production : Ces tests sont les différentes versions qu'on a envoyé à des bêta-testeur, pour avoir un retour et des bugs qu'on aurait pas vu.

APPLICATION

UNITAIRE

PRESENTATION

Les tests unitaires sont des tests effectué de manière « automatique » permettant de testé le code dans différents conditions et de testé le bon fonctionnement dans des cas normaux et des cas d'erreur.

Nous avons utilisé des tests unitaires pour les différentes classes se trouvant dans « components > model », voici les différentes classes testé en utilisant les tests unitaires :

- **User.ts** : La classe User.ts représente un utilisateur. Il permet de stocker toutes les données concernant un utilisateur.
- **Question.ts** : La classe Question.ts représente une question. Il permet de stocker toutes les données relatives à une question.
- **Player.ts :** La classe Player.ts représente un joueur dans une partie, ce qui n'est pas la même chose qu'un utilisateur. Elle permet de stocker les points récupérés pendant la partie.
- **Board.ts :** La classe Board représente un plateau dans une partie.

TEST EFFECTUES

Dans la partie ci-présentes nous allons vous présenter chaque méthode ayant été tester via des tests unitaires. Pour chaque méthode nous allons vous présenter une présentation de la méthode, le comportement attendu de la méthode, le comportement attendu de la méthode lorsqu'une erreur apparaît, et les différents tests réalisées sur cette méthode.

USER.TS :

- createUser :
  - **Présentation** : Méthode permettant de créer une instance basique d'un utilisateur, la méthode va prendre comme paramètre le pseudonyme de l'utilisateur, l'adresse mail de l'utilisateur et le mot de passe en clair de l'utilisateur.
  - **Comportement attendu** : La méthode doit générer un sel via PasswordUtil.getSalt(), puis il doit créer une instance d'un utilisateur avec le pseudo, le mail, le mot de passe en clair et le sel, les autres attributs sont initialisé à zéro.
  - **Comportement en cas d'erreur** : La méthode va se charger de créer l'instance de l'utilisateur, en cas d'erreur le constructeur se chargera de lancer l'exception.
  - **Test réalisées** : Création d'une instance utilisateur.
- createFullUser :
  - **Présentation** : Méthode permettant de créer une instance complète d'un utilisateur, la méthode va prendre comme paramètre tous les attributs de la classe et va instancier un utilisateur.
  - **Comportement attendu** : Instancier correctement un utilisateur.
  - **Comportement en cas d'erreur** : La méthode ne fait pas de test, le constructeur de User se chargera de lancer les exceptions.
  - **Test réalisées** : Création d'une instance complètes d'un utilisateur.
- validationPseudo :
  - **Présentation** : Création d'un utilisateur avec un pseudo valide, c'est-à-dire qui ne comporte pas d'espace et qui fait au minimum 4 et au maximum16 caractères.
  - **Comportement attendu** : Pour le pseudo ayant un espace le constructeur doit afficher une exceptions, pour le pseudo valide le constructeur doit créer une instance.
  - **Comportement en cas d'erreur :** Renvoyer une exception.

        - **Test réalisées** : Créer deux instances de User, le premier ayant un pseudo non valide comportant un espace, le second comportant un pseudo valide.
    - validationEmail :
        - **Présentation** : Le mail d'un utilisateur doit faire au maximum 64 caractères de long, le mail doit également suivre le paterne d'un mail classique c'est-à-dire le format « test@example.com »
        - **Comportement attendu** : Vérifier que le mail fait au plus 64 caractères et qu'il respecte le paterne.
        - **Comportement en cas d'erreur** : Si le mail fait plus de 64 caractères il doit renvoyé l'exception : « **Erreur : email trop long (plus de 64 caractères) !** », si le mail ne respecte pas le paterne il doit renvoyé l'exception : « **Erreur : email invalide !** »
        - **Test réalisées** : Créer trois instances de User, le premier ayant un mail non-valide ne comportant pas de caractère « @ », le second comportant un mail valide, le dernier ayant un mail de plus de 64 caractères de long.
    - setPoint :
        - **Présentation** : Un utilisateur doit toujours avoir un solde de point positifs ou égale à 0, il ne peut pas avoir un solde négatifs.
        - **Comportement attendu** : Accepter un solde de point positif ou égale à 0.
        - **Comportement en cas d'erreur** : Si solde négatifs renvoyé une exception.
        - **Test réalisées** : Création de deux instance de User, la première instancier avec un solde de point de 50, et la deuxième instancier avec un solde de point de -10.
    - setNewPassword :
        - **Présentation** : Permet de chiffré un mot de passe clair via PasswordUtil.hashPassword. On enregistre ensuite le mot de passe et le sel dans des attributs de l'instance de l'utilisateur.
        - **Comportement attendu** : Chiffré le mot de passe en générant un sel, puis en stockant tous dans les attributs de l'instance de l'utilisateur.
        - **Comportement en cas d'erreur** : Vérification que l'utilisateur fourni un mot de passe de plus de 10 caractères.
        - **Test réalisées** : Vérification du chiffrage et du stockage du mot de passe et du sel.

- setNbVictoire :
  - **Présentation** : Permet d'enregistrer le nombre de victoire de l'utilisateur.
  - **Comportement attendu** : Stocker le nombre de victoire dans l'attribut « **nombreVictoire** » de l'instance.
  - **Comportement en cas d'erreur** : Si le nombre de victoire est inférieure strictement à zéro alors une exception « **Le nombre de victoire ne doit pas être inferieur à 0** » se lance.
  - **Test réalisées** : On test une première fois avec un nombre de victoire positifs puis une autre fois avec un nombre de victoire négatifs
- setNbPartie :
  - **Présentation** : Permet d'enregistrer le nombre de partie de l'utilisateur.
  - **Comportement attendu** : Stocker le nombre de victoire dans l'attribut « **nombrePartie** » de l'instance.
  - **Comportement en cas d'erreur** : Si le nombre de partie est inférieure strictement à zéro alors une exception « **Le nombre de partie ne doit pas être inferieur à 0** » se lance.
  - **Test réalisées** : On test une première fois avec un nombre de partie positifs puis une autre fois avec un nombre de partie négatifs
- setPremium :
  - **Présentation** : Permet d'enregistrer l'état du compte utilisateur.
  - **Comportement attendu** : Permet de stocker des booléens comme états, si **True** alors l'utilisateur est premium sinon il n'est pas un utilisateur premium.
  - **Test réalisées** : On attribut un état à une instance de User.
- setPhotoProfil :
  - **Présentation** : Permet d'enregistrer la photo de profil de l'utilisateur dans l'attributs « **photoProfil** » de l'instance.
  - **Comportement attendu** : Récupérer l'id de la photo et la l'enregistrer dans l'attribut
  - **Comportement en cas d'erreur** : Si l'identifiant de la photo de profil est inferieur à zéro alors on lance l'exception « La photo de profil ne doit pas être inferieur à 0 ».
  - **Test réalisées** : On test avec deux photos de profil, une supérieur et une autre inférieur à zéro.

QUESTION.TS

La classe **question** permet uniquement de stocker des valeurs bruts dans des attributs pour qu'on puisse les récupérer lorsqu'on en à besoin. Elle contient uniquement un constructeur, puis des getters pour chaque champs.

Nous pouvons juste testé les méthodes « getters » pour voir si il retourne bien les bons champs.

- get id() :
    - **Présentation** : Retourne l'identifiant de la question.
    - **Comportement attendu** : Retourner le bon identifiant correspondant à la question.
    - **Test réalisées** : Création d'une instance de question avec l'identifiant qui est égale à 1. Nous vérifions ensuite que cette valeurs correspond avec celui qu'on obtient avec le getter.
- get typeReponse() :
    - **Présentation** : Retourne le type de réponse qu'on attend de l'utilisateur
    - **Comportement attendu** : Retourner le bon type pour pouvoir afficher correctement les boutons adéquats
    - **Test réalisées** : Création d'une instance de question avec le type de réponse qui est égale à « multiple choice ». Nous vérifions ensuite que cette valeurs correspond avec celui qu'on obtient avec le getter typeReponse.
- get correction() :
    - **Présentation** : Retourne la correction correspondant à la question.
    - **Comportement attendu** : Retourne la bonne correction qui correspond à la question.
    - **Test réalisées** : Création d'une instance de question avec la correction qui est « The correct answer is the sum of 2 and 2, which is 4». Nous vérifions ensuite que cette valeurs correspond avec celui qu'on obtient avec le getter correction.
- get question() :
    - **Présentation** : Retourne la question correspondant à l'instance.
    - **Comportement attendu** : Retourne la bonne question.
    - **Test réalisées** : Création d'une instance de question avec la question qui est « What is 2 + 2?». Nous vérifions ensuite que cette valeurs correspond avec celui qu'on obtient avec le getter question.

- get answer() :
    - **Présentation** : Retourne la réponse que l'application va attendre de l'utilisateur.
    - **Comportement attendu** : Retourne la bonne réponse dans le bon format.
    - **Test réalisées** : Création d'une instance de question avec la réponse qui est « 4 ». Nous vérifions ensuite que cette valeurs correspond avec celui qu'on obtient avec le getter answer.
- get category() :
    - **Présentation** : Retourne la catégorie de la question pour positionner correctement la question sur le plateau.
    - **Comportement attendu** : Retourne la bonne catégorie.
    - **Test réalisées** : Création d'une instance de question avec la catégorie qui est égale à Category.Calculs présent dans l'enumérateur Category.. Nous vérifions ensuite que cette valeurs correspond avec celui qu'on obtient avec le getter category.
- get img() :
    - **Présentation** : Retourne l'image de la question.
    - **Comportement attendu** : Retourne la bonne image.
    - **Test réalisées** : Création d'une instance de question avec l'image qui est égale à maths.png. Nous vérifions ensuite que cette valeurs correspond avec celui qu'on obtient avec le getter img.

PLAYER.TS

- Scan :
    - **Présentation** : Cette méthode permet de scanner les alentours du joueur pour savoir sur quelle case il peut allé selon son nombre sur le dé.
    - **Comportement attendu** : La méthode va faire tous les chemins et renvoyé les cases qui peuvent être atteinte sans revenir sur soi.
    - **Comportement en cas d'erreur :** Si aucune case n'est disponible, aucune case de ne sera retourné
    - **Test réalisées :** Voir si on a les bonnes case par rapport au numéro de dé

BOARD.TS :

- AddTile :
    - **Présentation :** La méthode principale pour créer un plateau, elle permet de créer une Tile (case) d'une catégorie de question (espace, calcul, etc.)
    - **Comportement attendu :** Créer une case et la relier avec ses voisin
    - **Comportement en cas d'erreur :**
    - **Test réalisées :** Créer une case dans un plateau de test et vérifier si elle a ses bon voisin

INTEGRATION

Les tests d'intégrations sont l'ensemble des tests qu'on a effectués pour vérifier le bon fonctionnement de toutes les parties du projets. Dans le projet MathsALaMaison nous avons 3 parties distinctes :

- L'application React-Native
- Le serveur Node JS
- La base de données MySQL

Chaque partie doit correctement faire son travail et chaque partie doit également pouvoir communiquer entre eux. L'application React-Native ne communique pas directement avec la base de données, il va le faire avec l'intermédiaire du serveur.

A chaque fois qu'une fonctionnalités avait été développé ont testé la connexion entre : l'application ←→ le serveur ←→ la base de données. Si une fonctionnalités ne concernait pas la connexion entre ces différents éléments alors ont testé uniquement l'impact que cela avait.

Si on ajouté une page classement alors on vérifié les autres modules qu'on importé, page de départ et les différentes pages relié à celle-ci. Donc à l'ajout de la page de classement on a testé :

- La page d'accueil
- Variable de session
- La classe ServerConnection.ts
- Le serveur
- La base de données

Une fois que les éléments ont été testé correctement, on pouvait passé au développement de la prochaine fonctionnalité.

IMPLEMENTATION

Les tests d'implémentation nous ont aidé à vérifier que le code qu'on a écrit faisait bien ce qu'on voulait.

Avant de développer une fonctionnalité on listé ce qu'on voulait que la fonctionnalité fassent, par exemple lorsqu'on développer connexion, on voulait que l'utilisateur puissent se connecter avec un identifiant (pseudonyme ou adresse mail) et son mot de passe. En faisant une liste de chose avant de coder nous a permis.

A chaque version notable, nous envoyons une version testable à différentes personnes pour avoir leur retour et les différents problèmes rencontrées.

Donc lorsqu'une personne de l'équipe avait finis de développé une fonctionnalité, il testait tous les cas qu'il pouvait. Lorsqu'on avait fait plusieurs fonctionnalités notables, on sortait une versions testables pour que les utilisateurs puisse les testés. Les utilisateurs pouvaient les testés via un QR Code qu'on leur envoyé.

Nous avons choisi les testeurs en fonction de l'appréhension technologies qu'ils avaient, nous avons choisi un étudiant en informatique (Mael COIGNARD en BUT1), et des personnes de métiers et âges différentes. Les testeurs était des personnes également avec un appareil différent des personnes de ceux du groupe, pour voir si l'affichage était correct.

SECURITE

Les tests de sécurités sont l'ensemble des tests qu'on a effectué pour protéger les données et l'application MathsALaMaison.

MOT DE PASSE

Dans un premier temps nous avons protéger les mots de passes des utilisateurs en les hachant. C'est-à-dire que nous ne stockons pas les mots de passe utilisateurs en clair. Lorsqu'un utilisateur veut définir son mot de passe, nous récupérons son mot de passe en clair, nous gênerons un sel via notre classe « password-util.ts », et nous générons un hash avec le module CryptoJS.

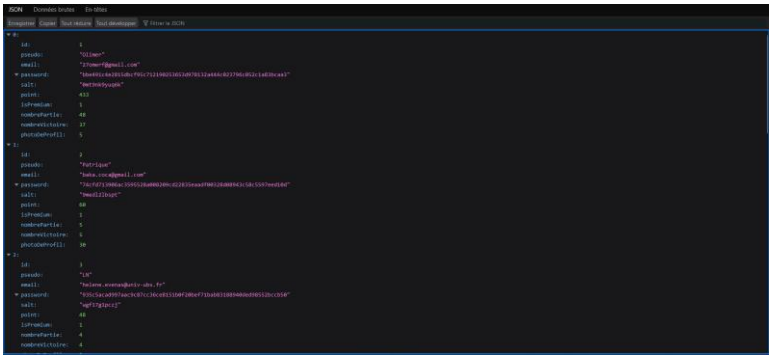
email	password	salt
27omerf@gmail.com	bbe491c4e2815dbcf95c712190253653d978132...	0mt9nk9yuq6k
baka.coca@gmail.com	74cfd713906ac3595528a008209cd22835eaadf...	9medl2lbspt
helene.evenas@univ-ubs.fr	935c5acad997aac9c87cc36ce8151b0f20bef71b...	wgf17g1pczj
romainperon29750@gmail.com	b7beb5df1eac4236101cabb4bcc8bb4d1c130fef...	lhskz4odhw
nono.parco@gmail.com	55324dc6b551761c1ee1b38f17a74ac38f36c5d...	1whqf13xloq

Lorsqu'un utilisateur veut se connecter nous récupérons le sel depuis la base de données puis nous hachons le mot de passe saisie, si le hash généré correspond à celui stocké dans la base de données alors on autorise la connexion.

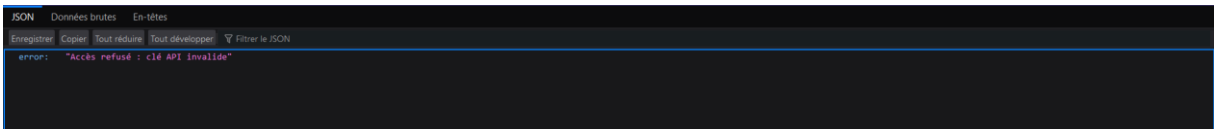
Avoir des mots de passes haché dans la base de données permettent même aux personnes ayant accès à la base de données de ne pas savoir les mots de passe des différents comptes utilisateurs.

CLE API

L'application MathsALaMaison communique avec un serveur externe, pour faire en sorte qu'uniquement l'application MathsALaMaison puisse communiquer avec le serveur, nous avons mis en place un système de token. Le serveur et l'application ont dans leurs code un token, qui va être envoyé dans le headers des différentes requêtes effectuée. Si le serveur ne reconnait pas le token, il ne va pas traiter les requêtes. Sans utiliser de token on peut accéder aux contenus de la base de données avec un navigateur.



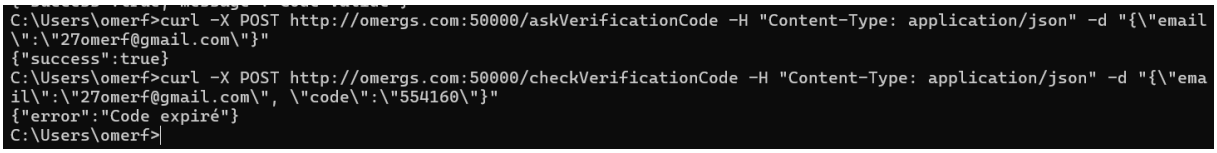
Après avoir ajouté le token, si on tente d'accéder via le navigateur on obtiendra ceci :



SERVEUR

ROUTES

Nous avons testé les différentes routes, via des requêtes de types GET ou/et des requêtes de types POST avec des outils comme CURL et l'application React-Native.



Ci-dessus on peut voir des tests pour vérifier le bon fonctionnement de la génération des codes de vérifications pour réinitialiser le mot de passe d'un compte utilisateur.

SECURITE

Pour communiquer avec le serveur l'application doit fournir un token, pour chaque requêtes entrantes on vérifie le token, si il est bon, on laisse passé la requêtes sinon on la bloque.