REPUBLIQUE DU CAMEROUN
*****
Paix-Travail-Patrie
*****
UNIVERSITE DE YAOUNDE I
****
FACULTE DES SCIENCES
****
DEPARTEMENT D'INFORMATIQUE
****

REPUBLIC OF CAMEROON
*****
Peace-Work-Fatherland
*****
THE UNIVERSITY OF YAOUNDE I
****
FACULTY OF SCIENCES
****
COMPUTER SCIENCE DEPARTMENT
****

Plan de test pour les tests unitaires

UE : INF352(Software Testing)

GROUPE :2

Matricule	Noms et Prénoms	Pourcentages
21T2650	NANA ORNELLA	100%
20U2743	BIYIHA MATIMBHE NOE MELODY	100%
20U816	DIGOU SENOU PRISNEL NICHA	100%
20U2851	DOMCHE WABO LYTHEVINE F.	100%

ENSEIGNANT

Dr KIMBI XAVERIA

TABLE DES MATIERES

Ce document de plan de test décrit les objectifs, la stratégie, les cas de test, et lles critères d'acceptation pour tester l'application qui nous a été donnée. Nos tests se concentrent sur cinq dossiers : basket, errorhandling,promotions,events,users.

I. Analyse des exigences

1. le dossier Basket les exigences sont

- Calculer le total du panier en fonction des articles ajoutés
- Appliquer des remises ou des promotions sur le total du panier.
- Afficher ou masquer les publicités en fonction du statut de l'utilisateur (premium ou non)
- Permettre la recherche d'articles dans le panier en fonction du nom de l'événement.
- Gérer l'ajout et la suppression d'articles dans le panier.
- Sérialiser les articles du panier en JSON pour les enregistrer ou les transmettre.

2. Le dossier error-handling les exigences sont

InvalidEventNameError :

- Cette erreur doit être levée lorsque le nom de l'événement n'est pas valide
- Elle doit hériter de la classe Error et avoir un constructeur qui prend un message d'erreur en paramètre.

InvalidEventPriceError :

- Cette erreur doit être levée lorsque le prix de l'événement n'est pas valide.
- Elle doit hériter de la classe Error et avoir un constructeur qui prend un message d'erreur en paramètre.

2. **InvalidUsernameError** :
   - Cette erreur doit être levée lorsque le nom d'utilisateur n'est pas valide.
   - Elle doit hériter de la classe Error et avoir un constructeur qui prend un message d'erreur en paramètre.

   InvalidReferralCodeError :
   - Cette erreur doit être levée lorsque le code de parrainage n'est pas valide.

- ➉ Elle doit hériter de la classe Error et avoir un constructeur qui prend un message d'erreur en paramètre.

3. **UserHasAccountError** :
   - ➉ Cette erreur doit être levée lorsqu'un utilisateur essaie de créer un compte alors qu'il en a déjà un.
   - ➉ Elle doit hériter de la classe Error et avoir un constructeur qui prend un message d'erreur en paramètre.

3. Le dossier event

le fichier event.js :

Classe Event :

- ➉ La classe Event doit avoir les propriétés suivantes :
- ➉ id : un identifiant unique de l'événement (peut être null lors de la création)
- ➉ name : le nom de l'événement
- ➉ ticketPrice: le prix des billets pour l'événement
- ➉ totalTickets: le nombre total de billets disponibles pour l'événement
- ➉ ticketsRemaining:: le nombre de billets restants pour l'événemen
- ➉ date : la date de l'événement
- ➉ Le constructeur de la classe Event doit initialiser ces propriétés.

2. **Fonction** isSoldOut :
   - ➉ Cette fonction doit prendre un objet Event en paramètre et retourner true si tous les billets sont vendus (c'est-à-dire si ticketsRemaining est égal à 0), et false sinon.
3. **Fonction** getTagLine :
   - ➉ Cette fonction doit prendre trois paramètres :
     - ➉ event : un objet Event
     - ➉ minimumTicketCount : un nombre entier représentant le nombre minimum de billets restants pour déclencher un message spécial
     - ➉ isPopular : un booléen indiquant si l'événement est populaire ou non
   - ➉ La fonction doit retourner une chaîne de caractères contenant un message de statut de l'événement, en fonction des conditions suivantes :
     - ➉ Si tous les billets sont vendus, retourner "Event Sold Out!"
     - ➉ Si le nombre de billets restants est inférieur à minimumTicketCount, retourner un message indiquant le nombre de billets restants

- Sinon, si isPopular est true, retourner un message indiquant que l'événement suscite un grand intérêt
- Sinon, retourner un message invitant à acheter des billets

4. **Fonction** createEvent :
   - Cette fonction doit prendre trois paramètres :
     - name : le nom de l'événement
     - price : le prix des billets pour l'événement
     - availableTickets : le nombre de billets disponibles pour l'événement
   - La fonction doit effectuer les validations suivantes :
     - Si name n'est pas une chaîne de caractères ou si sa longueur dépasse 200 caractères, lever une InvalidEventNameError.
     - Si price n'est pas un nombre ou est inférieur à 0, lever une InvalidEventPriceError.
     - Si availableTickets n'est pas un nombre ou est inférieur à 1, lever une InvalidEventPriceError.
   - Si toutes les validations sont passées, la fonction doit retourner un nouvel objet Event avec les valeurs fournies en paramètre.

Les fichiers filter.js et search .js

Fonction today :

- Cette fonction doit prendre un objet Event en paramètre et retourner true si la date de l'événement est égale à la date du jour, et falsesinon

2. **Fonction** next7Days :
   - Cette fonction doit prendre un objet Event en paramètre et retourner true si la date de l'événement se situe entre aujourd'hui et les 7 jours à venir (inclus), et false sinon.
3. **Fonction** next30Days :
   - Cette fonction doit prendre un objet Event en paramètre et retourner true si la date de l'événement se situe entre aujourd'hui et les 30 jours à venir (inclus), et false sinon.
1. **Fonction** getEvents :
   - Cette fonction doit prendre deux paramètres :
     - events : un tableau d'objets Event

- ❿ searchPredicate : une fonction de filtrage qui prend un objet Event en paramètre et retourne un booléen indiquant si l'événement doit être inclus ou non dans le résultat
- ❿ La fonction doit retourner un nouveau tableau contenant uniquement les événements qui répondent aux critères de filtrage définis par searchPredicate.

4. Le dossier promotions

discount.js

Le projet fournit une API pour gérer les réductions.

les réductions peuvent être en pourcentage ou en montant.

Les réductions ont un code de réduction unique.

exchange.test.js

- Le projet fournit une API pour obtenir les taux de change.

- Les taux de change sont disponibles pour différentes devises.

promotions.test.js

Spécifications du projet :

- Le projet fournit une interface pour gérer les promotions.

- Les promotions peuvent être en pourcentage ou en montant.

- Les promotions ont un code de promotion unique.

5. Le dossier Users

user.js

Définir une classe User avec les propriétés id, username et isPremium.

1. Implémenter une fonction userExists(username) qui vérifie si un utilisateur avec le nom d'utilisateur donné existe déjà.

2. Implémenter une fonction createUserId() qui génère un nouvel identifiant unique pour un nouvel utilisateur.

Account.js

1. Classe `Purchase

- Représente une transaction d'achat d'un utilisateur

Propriétés :

- `eventName` : nom de l'événement acheté

- `tickets` : nombre de billets achetés

- `cost` : coût total de l'achat

2. Fonction `isValidUserName(userName)`

- Vérifie si un nom d'utilisateur donné est valide

- Retourne `true` si le nom d'utilisateur est valide, `false` sinon

- Dans cette implémentation, un nom d'utilisateur est considéré comme valide s'il n'est pas vide et contient un caractère `@`

3. Fonction `createAccount(username)`

- Crée un nouveau compte utilisateur avec le nom d'utilisateur fourni

- Vérifie d'abord si le nom d'utilisateur est valide en appelant `isValidUserName(username)`

- Vérifie ensuite si l'utilisateur existe déjà en appelant `users.userExists(username)`

- Si l'utilisateur n'existe pas, crée un nouvel utilisateur en appelant `users.createUserId()` pour générer un nouvel identifiant unique

- Retourne un objet avec les données du nouvel utilisateur (`userId` et `username`)

- Si l'utilisateur existe déjà, rejette la promesse avec le message "User already exists

4. Fonction `getPastPurchases(userId)`

- Récupère l'historique des achats d'un utilisateur identifié par son `userId`

- Appelle `purchaseHistory.getPurchaseHistory(userId)` pour récupérer l'historique des achats

- Vérifie si la requête est terminée (`.readyState === 4`)

- Si la requête a réussi, retourne les événements achetés

- Sinon, lève une erreur avec le message "Failed to get purchase history"

purchaseHistory.js

getPurchaseHistory(userId) crée la requête HTTP pour récupérer les données d'historique des achats à partir de l'API.

parsePurchaseResponse(purchaseData) prend les données brutes de l'historique des achats et les transforme en un tableau d'objets Purchase.

II. Objectifs des tests

Ces objectifs de test permettent de s'assurer que les fonctionnalités spécifiques

des fichiers sont testés de manière efficace. Pour ce faire chaque fonction sera testé de façon isolé pour s ‘assurer qu’elle répondent aux exigences sus-cités nous appliquerons différentes techniques de test pour atteindre les objectifs suivant.

- Avoir une couverture maximale au moins 95% de notre code source.
- s’assurer que les fonction marchent bien autant pour des entrées valides que pour des entrées invalides.
- Minimiser les défauts et les bogues identifiés dans le code.
- Réduire le temps nécessaire pour exécuter les tests et les réparations.
- Améliorer la fiabilité de notre application

III. Définition de la portée des test

Nous effectuerons des tests unitaires qui vont couvrir la totalité des fonctions qui sont dans ces cinq dossiers. Sur chaque fonction seront appliqués des suites de tests pour s ‘assurer qu’elle réponde correctement aux exigences et aux critères de performance .

IV. Fonctionnalités hors portées

Nous définissons ici des aspects importants qui doivent être testés mais qui sont en dehors de la portée des tests unitaires. Ils sont généralement couverts par d'autres types de tests .

Tests d'intégration: Vérifier le comportement des fonctions lorsqu'elles sont utilisées dans le contexte de l'application complète. S'assurer que les fonctions interagissent correctement avec d'autres composants de l'application (par exemple, la récupération des événements depuis une API).

Tests de performance: Évaluer les performances des fonctions lorsqu'elles sont soumises à une charge importante (par exemple, un grand nombre d'événements). Identifier et résoudre les éventuels problèmes de performance.

Tests de compatibilité: Vérifier que les fonctions fonctionnent correctement sur différents navigateurs, systèmes d'exploitation et appareils. S'assurer que les fonctions sont compatibles avec les versions futures de l'application.

Tests de maintenance:Évaluer la facilité de modification et de mise à jour des fonctions.Vérifier que les changements apportés n'introduisent pas de régressions.

V. Stratégie de Test:

- Utiliser le framework de test Vitest pour ecrire des tets suites et les executer automatiquement .
- Utiliser les mocks pour simuler les interactions lorsqu'il y a des fonctions qui dépendent d'autres fonctions
- utiliser des mocks pour simuler les interactions avec axios.get.
- Écrire des tests unitaires pour chaque fonction.
- Exécuter les tests régulièrement pour détecter les régressions.
- Faire un rapport des tests pour voir ce qui doit être amélioré dans notre code ou modifié pour avoir de meilleures performances .

VI. Description des cas de tets

Pour chaque dossier de notre projet nous allons ecrire des tests case qui vont nous permettre de couvrir toutes les fonctions on appliquera des positives and negatives testing, des tests de valeur limites, des parametise tests….

⓾ Le dossier user

Fichier : `purchaseHistory.test.js

Test case1

Description : Vérifier que la fonction `getPurchaseHistory` retourne un objet `XMLHttpRequest` avec l'URL de requête correcte

- Arrange: Définir un ID d'utilisateur `'user123'`.

-Act: Appeler la fonction `getPurchaseHistory(userId)`.

Assert : Vérifier que l'objet retourné est une instance de `XMLHttpRequete . Vérifier que la méthode `open` a été appelée avec les paramètres `
'GET`'/account/orders/history?userId=user123'`.

TestCase2

Description : Vérifier que la fonction `parsePurchaseResponse` retourne un tableau d'objets `Purchase` avec les bonnes propriétés

-Arrange : Définir un jeu de données de test représentant les achats d'un utilisateur.

- Act : Appeler la fonction `parsePurchaseResponse(purchaseData)`.

- Assert : Vérifier que le résultat est un tableau d'objets.

Vérifier que le tableau a la bonne longueur (3 éléments).

Vérifier que chaque élément du tableau est une instance de `Purchase` et que ses propriétés sont correctes.

Test case3

Description : Vérifier que la fonction `getPurchaseHistory` retourne un objet de réponse avec la structure attendue, y compris les événements**

-Arrange : Aucune étape d'arrangement nécessaire.

-Act : Appeler la fonction `getPurchaseHistory()`.

- Assert :

- Vérifier que l'objet de réponse a les propriétés attendues (`readyState`, `onreadystatechange`, `response`).

- Vérifier que la propriété `response` a la structure attendue, avec un tableau d'événements.

- Vérifier que les données des événements sont correctes (noms, nombres de billets, prix).

Fichier : account.test.js

test case1

Description : Vérifier que la fonction `isValidUserName` retourne `false` pour un nom d'utilisateur vide

Arrange:

- Aucune étape d'arrangement nécessaire.

- Act :

- Appeler la fonction `isValidUserName` avec un nom d'utilisateur vide `"`.

- Assert :

- Vérifier que la fonction retourne `false`.

Test case2

Description : Vérifier que la fonction `isValidUserName` retourne `false` pour un nom d'utilisateur sans "@"

- Arrange: Aucune étape d'arrangement nécessaire.

- Act: Appeler la fonction `isValidUserName` avec un nom d'utilisateur sans "@" `'testuser'`.

- Assert : Vérifier que la fonction retourne `false`.

Test case3

Description : Vérifier que la fonction `isValidUserName` retourne `true` pour un nom d'utilisateur valide

- Arrange : Aucune étape d'arrangement nécessaire.

- Act : Appeler la fonction `isValidUserName` avec un nom d'utilisateur valide `'test@example.com'`.

- Assert: Vérifier que la fonction retourne `true`.

Testcase 4

Description : Vérifier que la fonction `createAccount` crée un nouveau compte avec un nom d'utilisateur valide**

- Arrange : Mocker la fonction `isValidUserName` pour qu'elle retourne `true`.

- Act : Appeler la fonction `createAccount` avec un nom d'utilisateur valide `'testUser@example.com'`.

- Assert: Vérifier que le résultat de `createAccount` est un objet avec la structure attendue `{ data: { userId: 2, username: 'testUser@example.com' } }`.

testcase5

Description : Vérifier que la fonction `createAccount` lève une erreur si le nom d'utilisateur est invalide

- Arrange: Mocker la fonction `isValidUserName` pour qu'elle retourne `false`.

- Act:

- Appeler la fonction `createAccount` avec un nom d'utilisateur invalide `'invalidUsername'`.

- Assert :

- Vérifier que la fonction `createAccount` rejette avec l'erreur `'Please enter a valid username'`.

- Vérifier que la fonction `isValidUserName` a été appelée avec le nom d'utilisateur `'invalidUsername'`.

- Vérifier que la fonction `userExists` n'a pas été appelée (car le nom d'utilisateur est invalide).

Testscase

Description : Vérifier que la fonction `createAccount` lève une erreur si l'utilisateur existe déjà

- Arrange : Mocker la fonction `userExists` pour qu'elle retourne `true` (l'utilisateur existe déjà).

- Act: Appeler la fonction `createAccount` avec un nom d'utilisateur existant `'existingUser@example.com'`.

-Assert

Vérifier que la fonction `createAccount` rejette avec l'erreur `'User already exists'`.

- Vérifier que la fonction `userExists` a été appelée avec le nom d'utilisateur `'existingUser@example.com'`.

Voici l'explication des tests cases pour le code fourni

Fichier : users.test.jq

Testcase1

Description : Vérifier que la fonction `userExists` retourne `true` pour un utilisateur existant

- Arrange: Aucune étape d'arrangement nécessaire.

- Act : Appeler la fonction `userExists` avec un utilisateur existant `'newuser1@pluralsight.com'`.

- Assert :Vérifier que la fonction retourne `true`.

Testcase2

Description : Vérifier que la fonction `userExists` retourne `false` pour un utilisateur non existant

- Arrange:Aucune étape d'arrangement nécessaire.

- Act : Appeler la fonction `userExists` avec un utilisateur non existant `'nonuser@example.com'`.

- Assert : Vérifier que la fonction retourne `false`.

Testcase3

Description : Vérifier que la fonction `userExists` gère correctement plusieurs utilisateurs existants

- Arrange :

- Aucune étape d'arrangement nécessaire.

- Act :Appeler la fonction `userExists` avec plusieurs utilisateurs existants `['newuser1@pluralsight.com', 'newuser1@pluralsight.com']`.

- Assert :Vérifier que la fonction retourne `true` pour chaque utilisateur

Testcase4

Description : Vérifier que la fonction `userExists` gère correctement plusieurs utilisateurs non existants

- Arrange :Aucune étape d'arrangement nécessaire.

- Act : Appeler la fonction `userExists` avec plusieurs utilisateurs non existants `['nonuser1@example.com', 'nonuser2@example.com']`.

- Assert: Vérifier que la fonction retourne `false` pour chaque utilisateur.

Testcase5

Description : Vérifier que la fonction `userExists` retourne `false` pour un tableau vide d'utilisateurs

- Arrange:

- Aucune étape d'arrangement nécessaire.

- Act: Appeler la fonction `userExists` avec un tableau vide d'utilisateurs `[]`.

- Assert: Vérifier que la fonction retourne `false` pour chaque utilisateur.

Testcase6

Description : Vérifier que la fonction `createUserId` retourne un entier positif**

- Arrange:

- Aucune étape d'arrangement nécessaire.

- Act: Appeler la fonction `createUserId`.

-Assert: Vérifier que le résultat est un entier positif.

Testcase7

Description : Vérifier que la fonction `createUserId` retourne un identifiant unique à chaque appel

- Arrange: Aucune étape d'arrangement nécessaire.

- Act : Appeler la fonction `createUserId` deux fois.

- Assert: Vérifier que les deux identifiants retournés sont différents

Testcase8

Description : Vérifier que la fonction `createUserId` retourne des identifiants différents dans un court délai

-Arrange : Aucune étape d'arrangement nécessaire.

- Act: Appeler la fonction `createUserId`, attendre 10 millisecondes, puis appeler à nouveau la fonction.

- Assert :Vérifier que les deux identifiants retournés sont différents.

Testcase9

Description : Vérifier que la fonction `createUserId` gère correctement les appels concurrents**

- Arrange : Aucune étape d'arrangement nécessaire.

-Act :Appeler la fonction `createUserId` de manière concurrente avec `Promise.all`.

- Assert :Vérifier que les deux identifiants retournés sont différents

Testcase10

Description : Vérifier que la fonction `createUserId` retourne le même identifiant lors d'appels successifs

- Arrange :Aucune étape d'arrangement nécessaire.

- Act :Appeler la fonction `createUserId`, puis l'appeler à nouveau.

- Assert :Vérifier que les deux identifiants retournés sont identiques.

- ❿ Le dossier promotions
- ❿ le dossier basket

  TC-001 : vérifie que la fonction `calculateTotal()` retourne bien 0 lorsque le panier est vide (aucun élément).

  TC-002 : vérifie que la fonction `calculateTotal()` calcule correctement le total lorsque le panier ne contient qu'un seul élément. Le résultat attendu est le prix de l'élément multiplié par sa quantité

TC-003 : Ce cas de test vérifie que la fonction `calculateTotal()` calcule correctement le total lorsque le panier contient plusieurs éléments. Le résultat attendu est la somme des prix de chaque élément multiplié par leur quantité respective

TC-004 : Ce cas de test vérifie que la fonction `calculateTotal()` applique correctement un rabais au total du panier. Le résultat attendu est la somme des prix de chaque élément multiplié par leur quantité respective, moins le montant du rabais.

TC-005 : Vérifier que les publicités ne s'affichent pas pour un utilisateur premium Ce cas de test vérifie que la fonction `showAdverts()` retourne `false` lorsque l'utilisateur est premium (a un abonnement premium).

TC-006 : Vérifier que les publicités s'affichent pour un utilisateur non-premium.Ce cas de test vérifie que la fonction `showAdverts()` retourne `true` lorsque l'utilisateur n'a pas d'abonnement premium.

cas de test pour les fonctions du dossier event

Fonction `isSoldOut` :

- Cas de test 1 : Vérifiez que la fonction retourne `true` lorsque `ticketsRemaining` est égal à 0.

- Cas de test 2 : Vérifiez que la fonction retourne `false` lorsque `ticketsRemaining` est négatif.

- Cas de test 3 : Vérifiez que la fonction retourne `false` lorsque `ticketsRemaining` est un nombre décimal.

- Cas de test 4 : Vérifiez que la fonction retourne `false` lorsque `ticketsRemaining` est une chaîne de caractères.

- Cas de test 5 : Vérifiez que la fonction retourne `false` lorsque `ticketsRemaining` est supérieur à 0.

Fonction getTagLine :

- Cas de test 1 : Vérifiez que la fonction retourne le message "Event Sold Out!" lorsque `isSoldOut` retourne `true`.

- Cas de test 2 : Vérifiez que la fonction retourne le message "Hurry only 5 tickets left!" lorsque `ticketsRemaining` est inférieur à `minimumTicketCount`.

- Cas de test 3 : Vérifiez que la fonction retourne le message "This Event is getting a lot of interest. Don't miss out, purchase your ticket now!" lorsque `isSoldOut` retourne `false` et `isPopular` retourne `true`.

- Cas de test 4 : Vérifiez que la fonction retourne le message "Don't miss out, purchase your ticket now!" lorsque `isSoldOut` retourne `false` et `isPopular` retourne `false`.

- Cas de test 5 : Vérifiez que la fonction retourne le message "Hurry only 10 tickets left!" lorsque `ticketsRemaining` est égal à `minimumTicketCount` et `isSoldOut` retourne `false`.

Fonction `createEvent` :

- Cas de test 1 : Vérifiez que la fonction lève une exception `InvalidEventNameError` lorsque le nom de l'événement est invalide (pas une chaîne de caractères).

- Cas de test 2 : Vérifiez que la fonction lève une exception `InvalidEventNameError` lorsque le nom de l'événement dépasse 200 caractères.

- Cas de test 3 : Vérifiez que la fonction lève une exception `InvalidEventPriceError` lorsque le prix de l'événement est invalide (négatif).

- Cas de test 4 : Vérifiez que la fonction lève une exception `InvalidEventPriceError` lorsque le prix de l'événement n'est pas un nombre.

- Cas de test 5 : Vérifiez que la fonction lève une exception `InvalidEventPriceError` lorsque `availableTickets` est inférieur à 1.

cas de test pour les fonctions du fichiers fiters.js

Plan de test pour le fichier `filters.js`:

Fonction `today`:

-cas de test 1 : Tester la fonction `today` avec une date dans le futur pour vérifier que la fonction renvoie `false`.

-cas de test 2: Tester la fonction `today` avec une date dans le passé pour vérifier que la fonction renvoie `false`.

-cas de tests 3 : Tester la fonction `today` avec une date correspondant à la date actuelle et une heure différente pour vérifier que la fonction renvoie `true`.

-cas de test 4: Tester la fonction `today` avec une date correspondant à la date actuelle et la même heure pour vérifier que la fonction renvoie `true`.

-cas de test 5: Tester la fonction `today` avec une date correspondant à la date actuelle mais une année différente pour vérifier que la fonction renvoie `false`.

Fonction `next7Days`:

-cas de test 1: Tester la fonction `next7Days` avec une date dans les 7 prochains jours pour vérifier que la fonction renvoie `true`.

-cas de test 2: Tester la fonction `next7Days` avec une date dans le passé pour vérifier que la fonction renvoie `false`.

-cas de test 3: Tester la fonction `next7Days` avec une date correspondant à la date actuelle et un jour différent pour vérifier que la fonction renvoie `true`.

-cas de test 4: Tester la fonction `next7Days` avec une date correspondant à la date actuelle et le même jour pour vérifier que la fonction renvoie `true`.

-cas de test 5: Tester la fonction `next7Days` avec une date correspondant à la date actuelle mais une année différente pour vérifier que la fonction renvoie `false`.

Fonction `next30Days`:

-cas de test 1: Tester la fonction `next30Days` avec une date dans les 30 prochains jours pour vérifier que la fonction renvoie `true`.

-cas de test 2: Tester la fonction `next30Days` avec une date dans le passé pour vérifier que la fonction renvoie `false`.

-cas de test 3: Tester la fonction `next30Days` avec une date correspondant à la date actuelle et un jour différent pour vérifier que la fonction renvoie `true`.

-cas de test 4: Tester la fonction `next30Days` avec une date correspondant à la date actuelle et le même jour pour vérifier que la fonction renvoie `true`.

-cas de test5:Tester la fonction `next30Days` avec une date correspondant à la date actuelle mais une année différente pour vérifier que la fonction renvoie `false`

cas de test pour les fonctions du fichier search.js

Fonction `getEvents`:

- cas de test 1: Vérifier que la fonction renvoie un tableau d'événements pour les dates dans les 7 prochains jours.

- cas de test 2: Vérifier que la fonction renvoie un tableau d'événements pour une date spécifique.

- cas de test 3: Vérifier que la fonction exclut les événements pour une date spécifique.

- cas de test 4: Vérifier que la fonction renvoie un tableau d'événements pendant le week-end.