

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 TESTS POUR LES TESTS UNITAIRES

UE : INF352(Software Testing)

GROUPE 2

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

ENSEIGNANT

Dr KIMBI XAVERIA

ANNEE ACADEMIQUE: 2023 - 2024

TABLE DES MATIÈRES

I. Analyse des exigences

1. exigences du dossier basket.
- 2.exigences du dossier error-handling
- 3.exigences du dossier events
- 4.exigences du dossiers promotions
5. exigences du dossier users

II. Objectifs et portée des tests

1. objectifs des tests
- 2.définition de la portée des tests
- 3.fonctionnalités hors portées

V. stratégie de test

- 1 . Environnement de test
2. Répartition des responsabilités au sein de l'équipe de tests
3. estimation de la durée du projet de tests

VI. description des cas de test

- 1.Test cases pour le dossier basket
- 2.Test cases pour le dossier error-handling
- 3.Test cases pour le dossier events
- 4 .Test cases pour le dossier promotions
- 5 .Test cases pour le dossier users

Ce document de plan de test décrit les objectifs, la stratégie, les cas de test, et les 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.

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.

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

UserHasAccountError :

Cette erreur doit être levée lorsqu'un utilisateur essaie de créer un compte alors qu'il en a déjà un.

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 :**

La classe **Event** doit avoir les propriétés suivantes :

id : un identifiant unique de l'événement (peut être **null** lors de la création)

name : le nom de l'événement

ticketPrice: le prix des billets pour l'événement

totalTickets: le nombre total de billets disponibles pour l'événement

ticketsRemaining:: le nombre de billets restants pour l'événement

date : la date de l'événement

Le constructeur de la classe **Event** doit initialiser ces propriétés.

Fonction **isSoldOut :**

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.

Fonction **getTagLine :**

Cette fonction doit prendre trois paramètres :

event : un objet **Event**

minimumTicketCount : un nombre entier représentant le nombre minimum de billets restants pour déclencher un message spécial

isPopular : un booléen indiquant si l'événement est populaire ou non

La fonction doit retourner une chaîne de caractères contenant un message de statut de l'événement, en fonction des conditions suivantes :

Si tous les billets sont vendus, retourner "Event Sold Out!"

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

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 `false` sinon

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.

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.

Fonction `getEvents` :

Cette fonction doit prendre deux paramètres :

`events` : un tableau d'objets `Event`

`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

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 et portée des tests

1. objectifs des tests

Ces objectifs de test permettent de s'assurer que les fonctionnalités spécifiques des fichiers sont testées de manière efficace. Pour ce faire chaque fonction sera testée de façon isolée pour s'assurer qu'elle répond aux exigences sus-citées. Nous appliquerons différentes techniques de test pour atteindre les objectifs suivants.

Avoir une **couverture maximale d'au moins 95%** de notre code source.

s'assurer que les fonctions 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

2. 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 .

3. 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

→ Environnement de test

- Utiliser le **framework de test Vitest** pour écrire des tests suites et les exécuter 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 .

→ Répartition des responsabilités au sein l'équipe chargée des tests

Pour réaliser les tests le travail sera répartie comme dans le tableau suivant :

Nom membre	Tâche à effectuer	durée (en jours)
BIYIHA NOÉ	Comprendre les exigences pour le dossier basket et écrire les potentiels cas de tests pour ce dossier	1
BIYIHA NOÉ	Comprendre les exigences pour le dossier error-handling et écrire les potentiels cas de tests pour ce dossier	1
NANA ORNELLA	Comprendre les exigences pour les fichiers du dossier events et écrire les potentiels tests case	2
DOMCHE LYTHEVINE	Comprendre les exigences pour les fichiers du dossier promotions et écrire les potentiels tests case	2
DIGOU SENOU PRISNEL	Comprendre les exigences pour les fichiers du dossier users et écrire les potentiels tests case	2
DIGOU SENOU PRISNEL	Rédiger un test plan pour le projet de tests avec les tests plan de chaque dossier fait par tous les membres	1
BIYIHA NOÉ	Écrire et exécuter les tests cases définis pour le dossier basket	1
NANA ORNELLA	Écrire et exécuter les tests cases définis pour le dossier events	2
DOMCHE LYTHEVINE	Écrire et exécuter les tests cases définis pour le dossier promotions	2
DIGOU PRISNEL	Écrire et exécuter les tests cases définis pour le dossier users	2
BIYIHA NOÉ	Écrire et exécuter les tests cases définis pour le dossier error-handling	1
DOMCHE LYTHEVINE	rédiger un rapport de test pour tous les tests unitaires que nous avons fait	1

Avec ce tableau nous pouvons estimer la durée pour réaliser ce travail, le tableau suivant va récapituler les tâches et les durées car il y a beaucoup de tâches indépendantes .

→ Estimation de la durée du projet

Activité	durée	précédent
1-Compréhension des exigences et rédaction des tests cases	2	///
2-Rédaction du test plan	1	1
3-Exécution des cas de tests	2	2
4- rédaction du rapport de tests	1	3

Après la construction d'un diagramme pert on conclut que la durée pour finir le travail est de 6 jours. Cependant toutes les tâches sont sur le chemin critique car un retard sur une seule activité va affecter la date de fin du projet .

VI. Description des cas de tests

Pour chaque dossier de notre projet nous allons écrire 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 parametrisé tests.... nous regroupons les tests cases par fonctions , fichiers , et dossiers .

1. Les tests cases pour le dossier basket

le fichier basketitem.js

Cas de test pour la fonctionnalité de panier d'achat

Test Case	Description	Inputs	Expected Output
TC-001	Vérifie que la fonction 'calculateTotal()' retourne bien 0 lorsque le panier est vide (aucun élément).	Panier vide	0
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.	Panier avec un seul élément	Prix de l'élément × 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.	Panier avec plusieurs éléments	Somme des (prix × quantité)
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.	Panier avec rabais	(Somme des (prix × quantité)) - 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).	Utilisateur premium	false
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.	Utilisateur non-premium	true

Table 1: Cas de test pour le fichier Basketitem.js

2. Les tests cases pour le dossier error-handling

Erreur	Cas de test	Description	Entrée	Sortie attendue
'InvalidEventNameError'	Cas de test 1	Vérifier que l'erreur est levée avec le message d'erreur approprié	'throw new InvalidEventNameError('Invalid event name')'	Erreur avec le message "Invalid event name"
'InvalidEventPriceError'	Cas de test 1	Vérifier que l'erreur est levée avec le message d'erreur approprié	'throw new InvalidEventPriceError('Invalid event price')'	Erreur avec le message "Invalid event price"
'InvalidUsernameError'	Cas de test 1	Vérifier que l'erreur est levée avec le message d'erreur approprié	'throw new InvalidUsernameError('Invalid username')'	Erreur avec le message "Invalid username"
'InvalidReferralCodeError'	Cas de test 1	Vérifier que l'erreur est levée avec le message d'erreur approprié	'throw new InvalidReferralCodeError('Invalid referral code')'	Erreur avec le message "Invalid referral code"
'UserHasAccountError'	Cas de test 1	Vérifier que l'erreur est levée avec le message d'erreur approprié	'throw new UserHasAccountError('User already has an account')'	Erreur avec le message "User already has an account"

Table 4: Plan de test pour les fonctions du fichier error-handling

3. Les tests case pour le dossier events

le fichier events.js

Fonction	Cas de test	Description	Entrée	Sortie attendue
'isSoldOut'	Cas de test 1	Vérifier que la fonction retourne 'true' lorsque 'ticketsRemaining' est égal à 0	'ticketsRemaining = 0'	'true'
'isSoldOut'	Cas de test 2	Vérifier que la fonction retourne 'false' lorsque 'ticketsRemaining' est négatif	'ticketsRemaining = -5'	'false'
'isSoldOut'	Cas de test 3	Vérifier que la fonction retourne 'false' lorsque 'ticketsRemaining' est un nombre décimal	'ticketsRemaining = 3.5'	'false'
'isSoldOut'	Cas de test 4	Vérifier que la fonction retourne 'false' lorsque 'ticketsRemaining' est une chaîne de caractères	'ticketsRemaining = "10"'	'false'
'isSoldOut'	Cas de test 5	Vérifier que la fonction retourne 'false' lorsque 'ticketsRemaining' est supérieur à 0	'ticketsRemaining = 10'	'false'
'getTagLine'	Cas de test 1	Vérifier que la fonction retourne le message "Event Sold Out!" lorsque 'isSoldOut' retourne 'true'	'isSoldOut = true'	"Event Sold Out!"
'getTagLine'	Cas de test 2	Vérifier que la fonction retourne le message "Hurry only 5 tickets left!" lorsque 'ticketsRemaining' est inférieur à 'minimumTicketCount'	'ticketsRemaining = 4', 'minimumTicketCount = 5'	"Hurry only 5 tickets left!"
'getTagLine'	Cas de test 3	Vérifier 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'	'isSoldOut = false', 'isPopular = true'	"This Event is getting a lot of interest. Don't miss out, purchase your ticket now!"
'getTagLine'	Cas de test 4	Vérifier que la fonction retourne le message "Don't miss out, purchase your ticket now!" lorsque 'isSoldOut' retourne 'false' et 'isPopular' retourne 'false'	'isSoldOut = false', 'isPopular = false'	"Don't miss out, purchase your ticket now!"
'getTagLine'	Cas de test 5	Vérifier que la fonction retourne le message "Hurry only 10 tickets left!" lorsque 'ticketsRemaining' est égal à 'minimumTicketCount' et 'isSoldOut' retourne 'false'	'ticketsRemaining = 10', 'minimumTicketCount = 10', 'isSoldOut = false'	"Hurry only 10 tickets left!"
'createEvent'	Cas de test 1	Vérifier que la fonction lève une exception 'InvalidEventNameError' lorsque le nom de l'événement est invalide (pas une chaîne de caractères)	'eventName = 123'	Lève 'InvalidEventNameError'
'createEvent'	Cas de test 2	Vérifier que la fonction lève une exception 'InvalidEventNameError' lorsque le nom de l'événement dépasse 200 caractères	'eventName = "A very long event name that exceeds the maximum length of 200 characters"'	Lève 'InvalidEventNameError'

Nom	Cas de test	Description	Entrée	Sortie attendue
'createEvent'	Cas de test 3	Vérifier que la fonction lève une exception 'InvalidEvent-PriceError' lorsque le prix de l'événement est invalide (négatif)	'eventPrice = -10'	Lève 'InvalidEvent-PriceError'
'createEvent'	Cas de test 4	Vérifier que la fonction lève une exception 'InvalidEvent-PriceError' lorsque le prix de l'événement n'est pas un nombre	'eventPrice = "abc"'	Lève 'InvalidEvent-PriceError'
'createEvent'	Cas de test 5	Vérifier que la fonction lève une exception 'InvalidEvent-PriceError' lorsque 'availableTickets' est inférieur à 1	'availableTickets = 0'	Lève 'InvalidEvent-PriceError'

le fichier search.js

Fonction	Cas de test	Description	Entrée	Sortie attendue
'getEvents'	Cas de test 1	Vérifier que la fonction renvoie un tableau d'événements pour les dates dans les 7 prochains jours	'getEvents(new Date(2024, 5, 20))'	Tableau d'événements pour les dates du 2024-06-20 au 2024-06-26
'getEvents'	Cas de test 2	Vérifier que la fonction renvoie un tableau d'événements pour une date spécifique	'getEvents(new Date(2024, 5, 22))'	Tableau d'événements pour le 2024-06-22
'getEvents'	Cas de test 3	Vérifier que la fonction exclut les événements pour une date spécifique	'getEvents(new Date(2024, 5, 21), new Date(2024, 5, 21))'	Tableau d'événements sans ceux du 2024-06-21
'getEvents'	Cas de test 4	Vérifier que la fonction renvoie un tableau d'événements pendant le week-end	'getEvents(new Date(2024, 5, 22))'	Tableau d'événements pour le samedi 2024-06-22 et le dimanche 2024-06-23

Table 3: Plan de test pour le fichier search.js

le fichier filter.js

Fonction	Cas de test	Description	Entrée	Sortie attendue
'today'	Cas de test 1	Vérifier que la fonction renvoie 'false' avec une date dans le futur	'date = new Date("2024-07-01")'	'false'
'today'	Cas de test 2	Vérifier que la fonction renvoie 'false' avec une date dans le passé	'date = new Date("2023-06-01")'	'false'
'today'	Cas de test 3	Vérifier que la fonction renvoie 'true' avec une date correspondant à la date actuelle, mais une heure différente	'date = new Date(2024, 5, 20, 12, 0, 0)'	'true'
'today'	Cas de test 4	Vérifier que la fonction renvoie 'true' avec une date correspondant à la date actuelle et la même heure	'date = new Date(2024, 5, 20, 0, 0, 0)'	'true'
'today'	Cas de test 5	Vérifier que la fonction renvoie 'false' avec une date correspondant à la date actuelle, mais une année différente	'date = new Date(2025, 5, 20, 0, 0, 0)'	'false'
'next7Days'	Cas de test 1	Vérifier que la fonction renvoie 'true' avec une date dans les 7 prochains jours	'date = new Date(2024, 5, 25)'	'true'
'next7Days'	Cas de test 2	Vérifier que la fonction renvoie 'false' avec une date dans le passé	'date = new Date(2023, 5, 15)'	'false'
'next7Days'	Cas de test 3	Vérifier que la fonction renvoie 'true' avec une date correspondant à la date actuelle et un jour différent	'date = new Date(2024, 5, 21)'	'true'
'next7Days'	Cas de test 4	Vérifier que la fonction renvoie 'true' avec une date correspondant à la date actuelle et le même jour	'date = new Date(2024, 5, 20)'	'true'
'next7Days'	Cas de test 5	Vérifier que la fonction renvoie 'false' avec une date correspondant à la date actuelle, mais une année différente	'date = new Date(2025, 5, 20)'	'false'
'next30Days'	Cas de test 1	Vérifier que la fonction renvoie 'true' avec une date dans les 30 prochains jours	'date = new Date(2024, 6, 10)'	'true'
'next30Days'	Cas de test 2	Vérifier que la fonction renvoie 'false' avec une date dans le passé	'date = new Date(2023, 5, 15)'	'false'
'next30Days'	Cas de test 3	Vérifier que la fonction renvoie 'true' avec une date correspondant à la date actuelle et un jour différent	'date = new Date(2024, 5, 30)'	'true'
'next30Days'	Cas de test 4	Vérifier que la fonction renvoie 'true' avec une date correspondant à la date actuelle et le même jour	'date = new Date(2024, 5, 20)'	'true'
'next30Days'	Cas de test 5	Vérifier que la fonction renvoie 'false' avec une date correspondant à la date actuelle, mais une année différente	'date = new Date(2025, 5, 20)'	'false'

Table 2: Plan de test pour les fonctions du fichier filters.js

3. Les tests cases pour le dossier promotions

le fichier discount.js

→ **discount.test.js**

ID	Description	Entrées	Sorties	Résultat attendu
1	getDiscount avec code non-string	12345	Erreur avec message "code must be a string"	Passé
2	getDiscount avec axios.get réussi	'ABC123'	Erreur avec message "code must be a string"	Passé
3	getDiscount avec axios.get réussi	'ABC123'	Renvoie les données de la réponse	Passé
4	getDiscount avec axios.get échoué	'ABC123'	Erreur avec message "Network Error"	Passé
5	getDiscount avec statut de réponse non-200	'ABC123'	Erreur avec message "Request failed with status code 404"	Passé

le fichier exchnage.js

→ exchange.test.js

ID	Description	Entrées	Sorties	Résultat attendu
1	getExchangeRate avec code de devise non-string	'USD'	Erreur avec message "Network Error"	Passé
2	getExchangeRate avec axios.get réussi	'USD'	Appelle axios.get avec l'URL '/exchange' et les paramètres { currencyCode: 'USD' }	Passé
3	getExchangeRate avec axios.get réussi	'USD'	Renvoie les données de la réponse	Passé
4	getExchangeRate avec axios.get échoué	'USD'	Erreur avec message "Network Error"	Passé
5	getExchangeRate avec statut de réponse non-200	'USD'	Erreur avec message "Request failed with status code 404"	Passé

Le fichier promotions.js

ID	Description	Entrées	Sortie attendue	Résultat attendu
1	Retourne la réduction correcte pour un prix actuel décimal	Pourcentage = 10, minimumSpend = 100, currentPrice = 150.50	$150.50 * (90 / 100) = 135.45$	La fonction retourne une valeur proche de 135.45 avec une précision de 2 décimales.
2	Retourne le prix actuel d'origine lorsque le prix actuel est inférieur au minimumSpend	Pourcentage = 10, minimumSpend = 100, currentPrice = 99.99	99.99	La fonction retourne la valeur 99.99
3	Retourne le prix réduit lorsque le prix actuel est égal au minimumSpend	Pourcentage = 10, minimumSpend = 100, currentPrice = 100	$100 * (90 / 100) = 90$	La fonction retourne la valeur 90.
4	Retourne le prix réduit lorsque le prix actuel est supérieur au minimumSpend et le pourcentage est 0	Pourcentage = 0, minimumSpend = 100, currentPrice = 150.50	150.50	La fonction retourne la valeur 150.50.
5	Retourne le prix réduit lorsque le prix actuel est supérieur au minimumSpend et le pourcentage est 100	Pourcentage = 100, minimumSpend = 100, currentPrice = 150.50	0	La fonction retourne la valeur 0.

5. Tests cases pour 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 `XMLHttpRequest` . 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 contient (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' } }`.

Test case5

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`

Fichier : users.test.

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.

Test case6

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.

Test case7

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

Test case8

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.

Test case9

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

Test case10

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.