Plan de test unitaire pour l'application mobile du restaurant

JULIE FONTSA DIANA 21Q2356,
GOUETH ANNE ALEXANDRA 22W2278,
YMELE TAKOUGANG DRADONE 21U2032,
KWETCHE FOKAM DARREL 21S2177,
LEKANE TATSA EPHRAIM 21T2868,
Superviseur : Mr RÉGIS ATEMENGUE

9 juin 2024

Contents

Objectif et Portée

Ce document a pour objectif de définir les étapes et les procédures de test unitaire pour l'application mobile du restaurant, en utilisant le framework Jest et suivant le modèle Arrange-Act-Assert (A-A-A). Les tests couvriront cinq sous-dossiers : basket, error-handling, events, promotions, et users.

Vue d'ensemble de l'application

L'application mobile permet aux utilisateurs de sélectionner des produits à partir d'un menu, de les ajouter à un panier, de consulter et d'appliquer des promotions, et de passer des commandes. Elle inclut également des fonctionnalités de gestion de profil et de suivi des commandes.

Objectifs des tests

Les objectifs des tests unitaires sont de :

- Vérifier que chaque fonction individuelle de l'application fonctionne correctement en isolation.
- Assurer que les composants de base sont fiables et robustes.
- Détecter et corriger les bogues précocement dans le cycle de développement.
- Faciliter les modifications futures en garantissant que les nouvelles modifications n'introduisent pas de régressions.

Approche des tests

Méthodes et techniques de test

- Utilisation de tests unitaires pour vérifier la logique des fonctions isolées.
- Adoption du modèle A-A-A pour structurer les tests de manière cohérente :
  - Arrange : Préparer les données et l'état initial nécessaires pour le test.
  - Act : Exécuter la fonction ou le module à tester.
  - Assert : Vérifier que le résultat obtenu correspond au résultat attendu.

Rôles et responsabilités

- Développeurs : Écrire et maintenir les tests unitaires.
- Testeurs : Exécuter les tests, analyser les résultats et signaler les bogues.
- Chef de projet de test : Superviser l'activité de test et assurer la qualité globale.

Calendrier des tests

Phase	Date de début	Date de fin
Planification	10/06/2024	15/06/2024
Conception	16/06/2024	20/06/2024
Écriture des tests	21/06/2024	30/06/2024
Exécution	01/07/2024	10/07/2024
Reporting	11/07/2024	15/07/2024

Environnement de test

Les tests unitaires seront exécutés dans l'environnement de développement utilisant :

- Framework de test : Jest.
- Configurations logicielles : Node.js, npm/yarn.
- Outils de CI/CD : GitHub Actions, Jenkins pour l'intégration continue.

Données de test

Les données de test incluront :

- Données simulées : Objets JSON représentant des utilisateurs, des produits, des promotions, etc.
- Mocking/Stubbing : Utilisation de bibliothèques comme jest.mock pour simuler des modules externes ou des dépendances.

Cas de test des sous-dossiers

Cas de test du sous-dossier Basket

0.1 Cas de test 1 : CalculateTotal_AvecEtSansDiscount

Description : Vérifier le calcul du total avec et sans remise
Hypothèses/Conditions préalables : Avoir une liste d'articles dans le panier
Tester les données :

- Deux articles dans le panier (Pizza Margherita à 10 € et Salade César à 7 €)
- Appliquer un discount de 10%

Étapes d'exécution :

1. Calculer le total sans discount
2. Calculer le total avec discount

Résultat :

- Total sans discount : 27 €
- Total avec discount : 26.9 €

Résultat et post-conditions : Test réussi

0.2 Cas de test 2 : ShowAdverts_PourUtilisateur

Description : Vérifier l'affichage des annonces en fonction du statut premium de l'utilisateur
Hypothèses/Conditions préalables : Avoir deux utilisateurs, l'un avec un abonnement premium, l'autre sans
Tester les données :

- Utilisateur avec abonnement premium
- Utilisateur sans abonnement premium

Étapes d'exécution :

1. Appeler la fonction showAdverts avec l'utilisateur premium
2. Appeler la fonction showAdverts avec l'utilisateur sans abonnement premium

Résultat :

- Pas d'affichage d'annonces pour l'utilisateur premium
- Affichage d'annonces pour l'utilisateur sans abonnement premium

Résultat et post-conditions : Test réussi

0.3 Cas de test 3 : SearchBasket_ArticlesCorrespondants

Description : Vérifier la recherche d'articles dans le panier
Hypothèses/Conditions préalables : Avoir une liste d'articles dans le panier
Tester les données :

- Articles dans le panier : Pizza Margherita, Salade César, Spaghetti Bolognese
- Requête de recherche : "Pizza"

Étapes d'exécution :

1. Appeler la fonction searchBasket avec la liste d'articles et la requête de recherche

Résultat :

- La recherche doit retourner 1 article correspondant : Pizza Margherita

Résultat et post-conditions : Test réussi

0.4 Cas de test 4 : GetBasketItem_ArticleSpecifique

Description : Vérifier la récupération d'un article spécifique dans le panier
Hypothèses/Conditions préalables : Avoir une liste d'articles dans le panier
Tester les données :

- Articles dans le panier : Pizza Margherita, Salade César
- Identification de l'article à récupérer : { id: 2 }

Étapes d'exécution :

1. Appeler la fonction getBasketItem avec la liste d'articles et l'identification de l'article

Résultat :

- L'article récupéré doit être la Salade César

Résultat et post-conditions : Test réussi

0.5 Cas de test 5 : CreateBasketItem_NouvelArticle

Description : Vérifier la création d'un nouvel article dans le panier
Hypothèses/Conditions préalables : Avoir une liste d'articles dans le panier
Tester les données :

- Articles existants dans le panier : Pizza Margherita, Salade César
- Nouvel article à créer : { name: 'Burger', id: 3, ticketPrice: 8 }, avec 2 billets requis

Étapes d'exécution :

1. Appeler la fonction createBasketItem avec la liste d'articles, le nouvel article et le nombre de billets requis

Résultat :

- Le nouvel article créé doit avoir le nom "Burger", l'ID 3 et le prix de 8, avec 2 billets

Résultat et post-conditions : Test réussi

0.6 Cas de test 6 : SerializeBasketItemsToJson_FormatCorrect

Description : Vérifier la sérialisation des articles du panier en format JSON
Hypothèses/Conditions préalables : Avoir une liste d'articles dans le panier
Tester les données :

- Articles dans le panier : Pizza Margherita (ID 1, prix 10 €, 2 billets), Salade César (ID 2, prix 7 €, 1 billet)

Étapes d'exécution :

1. Appeler la fonction serializeBasketItemsToJson avec la liste d'articles

Résultat :

- Le JSON généré doit correspondre à la structure attendue

Résultat et post-conditions : Test réussi

Cas de test du sous-dossier Events pour le fichier event.js

1 Cas de test 1 : CreateEvent_AvecEntreesValides

1.1 Étape 1 : ID du cas de test

CreateEvent_AvecEntreesValides

1.2 Étape 2 : Description du cas de test

Vérifier que la fonction createEvent() crée correctement un événement avec des entrées valides.

1.3 Étape 3 : Hypothèses/Conditions préalables

Aucune.

1.4 Étape 4 : Tester les données

- Nom de l'événement : "Concert de musique classique"
- Prix du billet : 50.00
- Nombre total de billets : 1000

1.5 Étape 5 : Étapes d'exécution

1. Appeler la fonction createEvent() avec les données de test.
2. Vérifier que l'événement créé a les bonnes valeurs pour les propriétés name, ticketPrice, totalTickets et ticketsRemaining.

1.6 Étape 6 : Résultat

L'événement créé a les valeurs attendues pour les propriétés suivantes :

- name = "Concert de musique classique"
- ticketPrice = 50.00
- totalTickets = 1000
- ticketsRemaining = 1000

1.7 Étape 7 : Résultat et post-conditions

Le test a réussi. L'événement a été créé correctement avec les entrées valides.

1.8 Étape 8 : Réussite ou échec

Réussite

2 Cas de test 2 : CreateEvent_AvecEntreesInvalides

2.1 Étape 1 : ID du cas de test

CreateEvent_AvecEntreesInvalides

2.2 Étape 2 : Description du cas de test

Vérifier que la fonction createEvent() lève les exceptions appropriées avec des entrées non valides.

2.3 Étape 3 : Hypothèses/Conditions préalables

Aucune.

2.4 Étape 4 : Tester les données

- Nom de l'événement invalide : 123
- Prix du billet invalide : -10.00
- Nombre total de billets invalide : 0

2.5 Étape 5 : Étapes d'exécution

1. Appeler la fonction createEvent() avec les données de test invalides.
2. Vérifier que les exceptions InvalidEventNameError et InvalidEventPriceError sont levées.

2.6 Étape 6 : Résultat

Les exceptions suivantes sont levées :

- Lors de l'appel avec un nom d'événement invalide (123) : InvalidEventNameError
- Lors de l'appel avec un prix de billet invalide (-10.00) : InvalidEventPriceError
- Lors de l'appel avec un nombre total de billets invalide (0) : InvalidEventPriceError

2.7 Étape 7 : Résultat et post-conditions

Le test a réussi. Les exceptions appropriées ont été levées avec les entrées non valides.

2.8 Étape 8 : Réussite ou échec

Réussite

3 Cas de test 3 : IsSoldOut_AvecEtSansTicketsRestants

3.1 Étape 1 : ID du cas de test

IsSoldOut_AvecEtSansTicketsRestants

3.2 Étape 2 : Description du cas de test

Vérifier que la fonction isSoldOut() retourne la bonne valeur pour des événements avec et sans billets restants.

3.3 Étape 3 : Hypothèses/Conditions préalables

Aucune.

3.4 Étape 4 : Tester les données

- Événement épuisé (0 billet restant)
- Événement avec des billets disponibles (50 billets restants)

3.5 Étape 5 : Étapes d'exécution

1. Créer les deux événements de test.
2. Appeler la fonction isSoldOut() pour chaque événement.
3. Vérifier que la fonction retourne true pour l'événement épuisé et false pour l'événement avec des billets disponibles.

3.6 Étape 6 : Résultat

- Pour l'événement épuisé (0 billet restant), isSoldOut() retourne true.
- Pour l'événement avec des billets disponibles (50 billets restants), isSoldOut() retourne false.

3.7 Étape 7 : Résultat et post-conditions

Le test a réussi. La fonction isSoldOut() a retourné les bonnes valeurs pour les deux cas de test.

3.8 Étape 8 : Réussite ou échec

Réussite

4 Cas de test 4 : GetTagLine_AvecDifferentsEtats

4.1 Étape 1 : ID du cas de test

GetTagLine_AvecDifferentsEtats

4.2 Étape 2 : Description du cas de test

Vérifier que la fonction getTagLine() retourne la bonne ligne d'information pour différents états d'événement.

4.3 Étape 3 : Hypothèses/Conditions préalables

- Le nombre minimum de billets restants pour être considéré comme populaire est de 10.

4.4 Étape 4 : Tester les données

- Événement épuisé (0 billet restant)
- Événement avec peu de billets restants (5 billets restants)
- Événement populaire (50 billets restants)
- Événement non populaire (50 billets restants)

4.5 Étape 5 : Étapes d'exécution

1. Créer les quatre événements de test.
2. Appeler la fonction getTagLine() pour chaque événement, en passant le nombre minimum de billets restants pour être considéré comme populaire.
3. Vérifier que la fonction retourne la ligne d'information attendue pour chaque événement.

4.6 Étape 6 : Résultat

- Pour l'événement épuisé (0 billet restant), getTagLine() retourne "Event Sold Out!".
- Pour l'événement avec peu de billets restants (5 billets restants), getTagLine() retourne "Hurry only 5 tickets left!".
- Pour l'événement populaire (50 billets restants), getTagLine() retourne "This Event is getting a lot of interest. Don't miss out, purchase your ticket now!".
- Pour l'événement non populaire (50 billets restants), getTagLine() retourne "Don't miss out, purchase your ticket now!".

4.7 Étape 7 : Résultat et post-conditions

Le test a réussi. La fonction getTagLine() a retourné les bonnes lignes d'information pour les différents états d'événement.

4.8 Étape 8 : Réussite ou échec

Réussite

Cas de test du sous-dossier Events pour le fichier filter.js

5 Cas de test : today_EventEnJour

5.1 Description

Vérifier que la fonction today() renvoie true lorsque l'événement a lieu aujourd'hui.

5.2 Hypothèses/Conditions préalables

L'événement a lieu aujourd'hui.

5.3 Tester les données

Une date correspondant à aujourd'hui.

5.4 Étapes d'exécution

1. Créer un événement avec une date correspondant à aujourd'hui.
2. Appeler la fonction today() avec l'événement en paramètre.

5.5 Résultat attendu

La fonction today() renvoie true.

5.6 Résultat et post-conditions

Le test réussit.

6 Cas de test : today_EventPasEnJour

6.1 Description

Vérifier que la fonction today() renvoie false lorsque l'événement n'a pas lieu aujourd'hui.

6.2 Hypothèses/Conditions préalables

L'événement n'a pas lieu aujourd'hui.

6.3 Tester les données

Une date correspondant à un jour après aujourd'hui.

6.4 Étapes d'exécution

1. Créer un événement avec une date correspondant à un jour après aujourd'hui.
2. Appeler la fonction today() avec l'événement en paramètre.

6.5 Résultat attendu

La fonction today() renvoie false.

6.6 Résultat et post-conditions

Le test réussit.

7 Cas de test : next7Days_EventDansLesProchains7Jours

7.1 Description

Vérifier que la fonction next7Days() renvoie true lorsque l'événement a lieu dans les 7 prochains jours.

7.2 Hypothèses/Conditions préalables

L'événement a lieu dans les 7 prochains jours.

7.3 Tester les données

Une date correspondant à 3 jours après aujourd'hui.

7.4 Étapes d'exécution

1. Créer un événement avec une date correspondant à 3 jours après aujourd'hui.
2. Appeler la fonction next7Days() avec l'événement en paramètre.

7.5 Résultat attendu

La fonction next7Days() renvoie true.

7.6 Résultat et post-conditions

Le test réussit.

8 Cas de test : next7Days_EventPasDansLesProchains7Jours

8.1 Description

Vérifier que la fonction next7Days() renvoie false lorsque l'événement n'a pas lieu dans les 7 prochains jours.

8.2 Hypothèses/Conditions préalables

L'événement n'a pas lieu dans les 7 prochains jours.

8.3 Tester les données

Une date correspondant à 8 jours après aujourd'hui.

8.4 Étapes d'exécution

1. Créer un événement avec une date correspondant à 8 jours après aujourd'hui.
2. Appeler la fonction next7Days() avec l'événement en paramètre.

8.5 Résultat attendu

La fonction next7Days() renvoie false.

8.6 Résultat et post-conditions

Le test réussit.

9 Cas de test : next30Days_EventDansLesProchains30Jours

9.1 Description

Vérifier que la fonction next30Days() renvoie true lorsque l'événement a lieu dans les 30 prochains jours.

9.2 Hypothèses/Conditions préalables

L'événement a lieu dans les 30 prochains jours.

9.3 Tester les données

Une date correspondant à 15 jours après aujourd'hui.

9.4 Étapes d'exécution

1. Créer un événement avec une date correspondant à 15 jours après aujourd'hui.
2. Appeler la fonction next30Days() avec l'événement en paramètre.

9.5 Résultat attendu

La fonction next30Days() renvoie true.

9.6 Résultat et post-conditions

Le test réussit.

10 Cas de test : next30Days_EventPasDansLesProchains30Jours

10.1 Description

Vérifier que la fonction next30Days() renvoie false lorsque l'événement n'a pas lieu dans les 30 prochains jours.

10.2 Hypothèses/Conditions préalables

L'événement n'a pas lieu dans les 30 prochains jours.

10.3 Tester les données

Une date correspondant à 31 jours après aujourd'hui.

10.4 Étapes d'exécution

1. Créer un événement avec une date correspondant à 31 jours après aujourd'hui.
2. Appeler la fonction next30Days() avec l'événement en paramètre.

10.5 Résultat attendu

La fonction next30Days() renvoie false.

10.6 Résultat et post-conditions

Le test réussit.

Cas de test du sous-dossier Events pour le fichier search.js

11 Cas de test 1 : Filtrer des événements avec une fonction de recherche valide

11.1 Description

Vérifier que la fonction getEvents() filtre correctement les événements en utilisant une fonction de recherche valide.

11.2 Hypothèses/Conditions préalables

Un tableau d'événements et une fonction de recherche valide sont fournis.

11.3 Tester les données

- Tableau d'événements : [new Event(1, 'Event 1', 100, 1000, 500, new Date('2023-06-01')), new Event(2, 'Event 2', 50, 500, 100, new Date('2023-06-15')), new Event(3, 'Event 3', 75, 750, 400, new Date('2023-06-01')),]
- Fonction de recherche valide : (event) ⇒ event.date.getDate() === 1

11.4 Étapes d'exécution

1. Appeler la fonction getEvents() avec le tableau d'événements et la fonction de recherche valide.
2. Vérifier que le tableau d'événements filtrés a une longueur inférieure ou égale à la longueur du tableau d'événements d'entrée.
3. Vérifier que le tableau d'événements filtrés contient les événements attendus.

11.5 Résultat

Le tableau d'événements filtrés contient les événements attendus.

11.6 Résultat et post-conditions

Succès. La fonction getEvents() a correctement filtré les événements en utilisant la fonction de recherche valide.

12 Cas de test 2 : Filtrer un tableau d'événements vide

12.1 Description

Vérifier que la fonction getEvents() retourne un tableau vide lorsqu'elle est appelée avec un tableau d'événements vide.

12.2 Hypothèses/Conditions préalables

Un tableau d'événements vide et une fonction de recherche valide sont fournis.

12.3 Tester les données

- Tableau d'événements : []
- Fonction de recherche valide : (event) ⇒ event.date.getDate() === 1

12.4 Étapes d'exécution

1. Appeler la fonction getEvents() avec le tableau d'événements vide et la fonction de recherche valide.
2. Vérifier que le tableau d'événements filtrés est vide.

12.5 Résultat

Le tableau d'événements filtrés est vide.

12.6 Résultat et post-conditions

Succès. La fonction getEvents() a correctement retourné un tableau vide lorsqu'elle a été appelée avec un tableau d'événements vide.

13 Cas de test 3 : Utiliser une fonction de recherche invalide

13.1 Description

Vérifier que la fonction getEvents() lève une erreur lorsqu'elle est appelée avec une fonction de recherche invalide.

13.2 Hypothèses/Conditions préalables

Un tableau d'événements et une fonction de recherche invalide sont fournis.

13.3 Tester les données

- Tableau d'événements : [new Event(1, 'Event 1', 100, 1000, 500, new Date('2023-06-01')), new Event(2, 'Event 2', 50, 500, 100, new Date('2023-06-15')), new Event(3, 'Event 3', 75, 750, 400, new Date('2023-06-01')),]
- Fonction de recherche invalide : `null`

13.4 Étapes d'exécution

1. Appeler la fonction getEvents() avec le tableau d'événements et la fonction de recherche invalide.
2. Vérifier que la fonction lève une erreur.

13.5 Résultat

La fonction getEvents() lève une erreur.

13.6 Résultat et post-conditions

Succès. La fonction getEvents() a correctement levé une erreur lorsqu'elle a été appelée avec une fonction de recherche invalide.

14 Cas de test 4 : Filtrer des événements avec une fonction de recherche qui retourne toujours true

14.1 Description

Vérifier que la fonction getEvents() retourne tous les événements lorsqu'elle est appelée avec une fonction de recherche qui retourne toujours true.

14.2 Hypothèses/Conditions préalables

Un tableau d'événements et une fonction de recherche qui retourne toujours true sont fournis.

14.3 Tester les données

- Tableau d'événements : [new Event(1, 'Event 1', 100, 1000, 500, new Date('2023-06-01')), new Event(2, 'Event 2', 50, 500, 100, new Date('2023-06-15')), new Event(3, 'Event 3', 75, 750, 400, new Date('2023-06-01')),]
- Fonction de recherche : ⇒ `true`

14.4 Étapes d'exécution

1. Appeler la fonction getEvents() avec le tableau d'événements et la fonction de recherche qui retourne toujours true.
2. Vérifier que le tableau d'événements filtrés contient tous les événements du tableau d'entrée.

14.5 Résultat

Le tableau d'événements filtrés contient tous les événements du tableau d'entrée.

14.6 Résultat et post-conditions

Succès. La fonction getEvents() a correctement retourné tous les événements lorsqu'elle a été appelée avec une fonction de recherche qui retourne toujours true.

Cas de test du sous-dossier Promotions/exchange

15 Cas de test 1

15.1 Étape 1 : ID du cas de test

TC01

15.2 Étape 2 : Description du cas de test

Récupération du taux de change pour une devise valide

15.3 Étape 3 : Hypothèses/Conditions préalables

La devise "USD" est valide

15.4 Étape 4 : Tester les données

Devise : "USD", Taux de change attendu : 1.25

15.5 Étape 5 : Étapes d'exécution

1. Définir le taux de change attendu
2. Configurer le mockage de l'appel à exchangeRateProvider.callExchangeRateProvider pour retourner le taux de change attendu
3. Appeler la fonction getExchangeRate avec la devise "USD" et un callback

4. Vérifier que la fonction exchangeRateProvider.callExchangeRateProvider a été appelée avec la devise "USD"

5. Vérifier que le callback a été appelé avec la réponse attendue

15.6 Étape 6 : Résultat

Le test passe

15.7 Étape 7 : Résultat et post-conditions

Le taux de change est correctement récupéré pour la devise valide "USD"

15.8 Étape 8 : Réussite

16 Cas de test 2

16.1 Étape 1 : ID du cas de test

TC02

16.2 Étape 2 : Description du cas de test

Gestion d'une devise non supportée

16.3 Étape 3 : Hypothèses/Conditions préalables

La devise "INVALID" n'est pas prise en charge

16.4 Étape 4 : Tester les données

Devise : "INVALID", Erreur attendue : "Currency not supported"

16.5 Étape 5 : Étapes d'exécution

1. Configurer le mockage de l'appel à exchangeRateProvider.callExchangeRateProvider pour lever une erreur "Currency not supported"

2. Appeler la fonction getExchangeRate avec la devise "INVALID" et un callback

3. Vérifier que la fonction exchangeRateProvider.callExchangeRateProvider a été appelée avec la devise "INVALID"

4. Vérifier que le callback a été appelé avec la réponse attendue contenant l'erreur

16.6 Étape 6 : Résultat

Le test passe

16.7 Étape 7 : Résultat et post-conditions

L'erreur "Currency not supported" est correctement gérée pour la devise non supportée "INVALID"

16.8 Étape 8 : Réussite

17 Cas de test 3

17.1 Étape 1 : ID du cas de test

TC03

17.2 Étape 2 : Description du cas de test

Appel du callback fourni

17.3 Étape 3 : Hypothèses/Conditions préalables

La devise "EUR" est valide

17.4 Étape 4 : Tester les données

Devise : "EUR", Taux de change attendu : 1.18

17.5 Étape 5 : Étapes d'exécution

1. Définir le taux de change attendu
2. Configurer le mockage de l'appel à exchangeRateProvider.callExchangeRateProvider pour retourner le taux de change attendu
3. Créer un mock pour le callback
4. Appeler la fonction getExchangeRate avec la devise "EUR" et le callback
5. Vérifier que la fonction exchangeRateProvider.callExchangeRateProvider a été appelée avec la devise "EUR"
6. Vérifier que le callback a été appelé avec la réponse attendue

17.6 Étape 6 : Résultat

Le test passe

17.7 Étape 7 : Résultat et post-conditions

Le callback fourni est bien appelé avec la réponse attendue

17.8 Étape 8 : Réussite

Cas de test du sous-dossier Promotions

18 Cas de test 1 : calculatePercentageDiscount_AvecPrixSupérieurAuMonta

18.1 Étape 1 : ID du cas de test

calculatePercentageDiscount_AvecPrixSupérieurAuMontantMinimum

18.2 Étape 2 : Description du cas de test

Vérifier que la fonction calculatePercentageDiscount calcule correctement le prix après remise lorsque le prix actuel est supérieur au montant minimum requis.

18.3 Étape 3 : Hypothèses/Conditions préalables

La remise est de 20%, le montant minimum requis est de 50 et le prix actuel est de 100.

18.4 Étape 4 : Tester les données

Appeler la fonction calculatePercentageDiscount avec les données ci-dessus.

18.5 Étape 5 : Étapes d'exécution

Appeler la fonction calculatePercentageDiscount avec les paramètres percentage = 20, minimumSpend = 50 et currentPrice = 100.

18.6 Étape 6 : Résultat

Le résultat attendu est 80.

18.7 Étape 7 : Résultat et post-conditions

Le résultat obtenu est 80, ce qui correspond au résultat attendu. Le test a donc réussi.

18.8 Étape 8 : Réussite ou échec

Réussite

19 Cas de test 2 : calculatePercentageDiscount_AvecPrixInférieurAuMontan

19.1 Étape 1 : ID du cas de test

calculatePercentageDiscount_AvecPrixInférieurAuMontantMinimum

19.2 Étape 2 : Description du cas de test

Vérifier que la fonction calculatePercentageDiscount calcule correctement le prix après remise lorsque le prix actuel est inférieur au montant minimum requis.

19.3 Étape 3 : Hypothèses/Conditions préalables

La remise est de 20%, le montant minimum requis est de 50 et le prix actuel est de 40.

19.4 Étape 4 : Tester les données

Appeler la fonction calculatePercentageDiscount avec les données ci-dessus.

19.5 Étape 5 : Étapes d'exécution

Appeler la fonction calculatePercentageDiscount avec les paramètres percentage = 20, minimumSpend = 50 et currentPrice = 40.

19.6 Étape 6 : Résultat

Le résultat attendu est 40.

19.7 Étape 7 : Résultat et post-conditions

Le résultat obtenu est 40, ce qui correspond au résultat attendu. Le test a donc réussi.

19.8 Étape 8 : Réussite ou échec

Réussite

20 Cas de test 3 : calculateMoneyOff_AvecPrixSupérieurAuMontantMinimu

20.1 Étape 1 : ID du cas de test

calculateMoneyOff_AvecPrixSupérieurAuMontantMinimum

20.2 Étape 2 : Description du cas de test

Vérifier que la fonction calculateMoneyOff calcule correctement le prix après remise lorsque le prix actuel est supérieur au montant minimum requis.

20.3 Étape 3 : Hypothèses/Conditions préalables

La remise est de 20, le montant minimum requis est de 50 et le prix actuel est de 100.

20.4 Étape 4 : Tester les données

Appeler la fonction calculateMoneyOff avec les données ci-dessus.

20.5 Étape 5 : Étapes d'exécution

Appeler la fonction calculateMoneyOff avec les paramètres discount = 20, minimumSpend = 50 et currentPrice = 100.

20.6 Étape 6 : Résultat

Le résultat attendu est 80.

20.7 Étape 7 : Résultat et post-conditions

Le résultat obtenu est 80, ce qui correspond au résultat attendu. Le test a donc réussi.

20.8 Étape 8 : Réussite ou échec

Réussite

21 Cas de test 4 : calculateMoneyOff_AvecPrixInférieurAuMontantMinimu

21.1 Étape 1 : ID du cas de test

calculateMoneyOff_AvecPrixInférieurAuMontantMinimum

21.2 Étape 2 : Description du cas de test

Vérifier que la fonction calculateMoneyOff calcule correctement le prix après remise lorsque le prix actuel est inférieur au montant minimum requis.

21.3 Étape 3 : Hypothèses/Conditions préalables

La remise est de 20, le montant minimum requis est de 50 et le prix actuel est de 40.

21.4 Étape 4 : Tester les données

Appeler la fonction calculateMoneyOff avec les données ci-dessus.

21.5 Étape 5 : Étapes d'exécution

Appeler la fonction calculateMoneyOff avec les paramètres discount = 20, minimumSpend = 50 et currentPrice = 40.

21.6 Étape 6 : Résultat

Le résultat attendu est 40.

21.7 Étape 7 : Résultat et post-conditions

Le résultat obtenu est 40, ce qui correspond au résultat attendu. Le test a donc réussi.

21.8 Étape 8 : Réussite ou échec

Réussite

22 Cas de test 5 : generateReferralCode_AvecIdUtilisateurValid

22.1 Étape 1 : ID du cas de test

generateReferralCode_AvecIdUtilisateurValid

22.2 Étape 2 : Description du cas de test

Vérifier que la fonction generateReferralCode génère un code de parrainage valide lorsque l'ID de l'utilisateur est valide.

22.3 Étape 3 : Hypothèses/Conditions préalables

L'ID de l'utilisateur est 123.

22.4 Étape 4 : Tester les données

Appeler la fonction generateReferralCode avec l'ID de l'utilisateur.

22.5 Étape 5 : Étapes d'exécution

Appeler la fonction generateReferralCode avec le paramètre userId = 123.

22.6 Étape 6 : Résultat

Le résultat attendu est un code de parrainage au format #FRIEND-#3-#123.

22.7 Étape 7 : Résultat et post-conditions

Le résultat obtenu correspond au format attendu. Le test a donc réussi.

22.8 Étape 8 : Réussite ou échec

Réussite

23 Cas de test 6 : applyDiscount_AvecCodeDeRemiseValide

23.1 Étape 1 : ID du cas de test

applyDiscount_AvecCodeDeRemiseValide

23.2 Étape 2 : Description du cas de test

Vérifier que la fonction applyDiscount applique correctement la remise lorsque le code de remise est valide.

23.3 Étape 3 : Hypothèses/Conditions préalables

Le code de remise est "DISCOUNT10", le montant total actuel est de 100, le code de remise est valide, le type de remise est "PERCENTAGEOFF" avec une valeur de 10% et un montant minimum requis de 50.

23.4 Étape 4 : Tester les données

Appeler la fonction applyDiscount avec le code de remise et le montant total actuel.

23.5 Étape 5 : Étapes d'exécution

Appeler la fonction applyDiscount avec les paramètres discountCode = 'DISCOUNT10' et currentTotal = 100.

23.6 Étape 6 : Résultat

Le résultat attendu est 90.

23.7 Étape 7 : Résultat et post-conditions

Le résultat obtenu est 90, ce qui correspond au résultat attendu. Le test a donc réussi.

23.8 Étape 8 : Réussite ou échec

Réussite

Cas de test du sous-dossier users/purchaseHistory

24 Cas de test 1

24.1 Étape 1 : Test de la fonction getPurchaseHistory

24.2 Étape 2 : Vérifier que la fonction retourne une promesse qui se résout en un tableau d'objets Purchase

24.3 Étape 3 : Avoir un utilisateur avec un ID valide et des achats dans l'historique

24.4 Étape 4 : Données de test : ID d'utilisateur '123' et des données d'achat fictives

24.5 Étape 5 :

1. Configurer le module Fetch pour simuler une réponse avec les données d'achat fictives
2. Appeler la fonction getPurchaseHistory avec l'ID d'utilisateur
3. Vérifier que la fonction Fetch a été appelée avec l'URL correcte
4. Vérifier que le résultat est un tableau d'objets Purchase correspondant aux données d'achat fictives

24.6 Étape 6 : Les tests unitaires passent

24.7 Étape 7 : La fonction getPurchaseHistory retourne correctement un tableau d'objets Purchase à partir des données d'achat

24.8 Étape 8 : Succès

25 Cas de test 2

25.1 Étape 1 : Test de la fonction parsePurchaseResponse

25.2 Étape 2 : Vérifier que la fonction transforme correctement les données d'achat en tableau d'objets Purchase

25.3 Étape 3 : Avoir des données d'achat valides

25.4 Étape 4 : Données de test : Tableau d'objets d'achat fictifs

25.5 Étape 5 :

1. Appeler la fonction parsePurchaseResponse avec les données d'achat fictives
2. Vérifier que le résultat est un tableau d'objets Purchase correspondant aux données d'achat fictives

25.6 Étape 6 : Les tests unitaires passent

25.7 Étape 7 : La fonction parsePurchaseResponse transforme correctement les données d'achat en tableau d'objets Purchase

25.8 Étape 8 : Succès

Cas de test du sous-dossier users/account

26 Cas de test 1

26.1 Étape 1 : Test de la fonction createAccount avec un nom d'utilisateur valide

26.2 Étape 2 : Vérifier que la fonction crée un compte avec un ID d'utilisateur et un nom d'utilisateur corrects

26.3 Étape 3 : L'utilisateur n'existe pas encore

26.4 Étape 4 : Nom d'utilisateur : "newuser@example.com"

26.5 Étape 5 :

1. Configurer le module users pour simuler que l'utilisateur n'existe pas
2. Appeler la fonction createAccount avec le nom d'utilisateur valide
3. Vérifier que le résultat correspond à l'attente (objet contenant l'ID et le nom d'utilisateur)

26.6 Étape 6 : Le test passe

26.7 Étape 7 : La fonction createAccount crée correctement un compte avec un ID et un nom d'utilisateur valides

26.8 Étape 8 : Succès

27 Cas de test 2

27.1 Étape 1 : Test de la fonction createAccount avec un nom d'utilisateur invalide

27.2 Étape 2 : Vérifier que la fonction lève une InvalidUsernameError

27.3 Étape 3 : Aucune condition préalable requise

27.4 Étape 4 : Nom d'utilisateur : "invaliduser.com"

27.5 Étape 5 :

1. Appeler la fonction createAccount avec le nom d'utilisateur invalide
2. Vérifier que la promesse est rejetée avec une InvalidUsernameError

27.6 Étape 6 : Le test passe

27.7 Étape 7 : La fonction createAccount lève correctement une InvalidUsernameError pour un nom d'utilisateur invalide

27.8 Étape 8 : Succès

28 Cas de test 3

28.1 Étape 1 : Test de la fonction createAccount avec un utilisateur existant

28.2 Étape 2 : Vérifier que la fonction rejette avec une erreur indiquant que l'utilisateur existe déjà

28.3 Étape 3 : L'utilisateur existe déjà

28.4 Étape 4 : Nom d'utilisateur : "existinguser@example.com"

28.5 Étape 5 :

1. Configurer le module users pour simuler que l'utilisateur existe déjà
2. Appeler la fonction createAccount avec le nom d'utilisateur existant
3. Vérifier que la promesse est rejetée avec une erreur indiquant que l'utilisateur existe déjà

28.6 Étape 6 : Le test passe

28.7 Étape 7 : La fonction createAccount rejette correctement avec une erreur indiquant que l'utilisateur existe déjà

28.8 Étape 8 : Succès

29 Cas de test 4

29.1 Étape 1 : Test de la fonction getPastPurchases avec une requête réussie

29.2 Étape 2 : Vérifier que la fonction retourne les achats passés

29.3 Étape 3 : L'utilisateur a des achats passés

29.4 Étape 4 : Aucune donnée de test requise

29.5 Étape 5 :

1. Configurer le module purchaseHistory pour simuler une réponse réussie avec des données d'achat
2. Appeler la fonction getPastPurchases
3. Vérifier que le résultat correspond aux données d'achat attendues

29.6 Étape 6 : Le test passe

29.7 Étape 7 : La fonction getPastPurchases retourne correctement les achats passés de l'utilisateur

29.8 Étape 8 : Succès

30 Cas de test 5

30.1 Étape 1 : Test de la fonction getPastPurchases avec une requête échouée

30.2 Étape 2 : Vérifier que la fonction lève une erreur lorsque la requête échoue

30.3 Étape 3 : Aucune condition préalable requise

30.4 Étape 4 : Aucune donnée de test requise

30.5 Étape 5 :

1. Configurer le module purchaseHistory pour simuler une erreur lors de la requête
2. Appeler la fonction getPastPurchases
3. Vérifier que la promesse est rejetée avec une erreur

30.6 Étape 6 : Le test passe

30.7 Étape 7 : La fonction getPastPurchases lève correctement une erreur lorsque la requête échoue

30.8 Étape 8 : Succès

Automatisation des tests

Les tests unitaires seront automatisés avec Jest et exécutés à chaque commit via un pipeline CI/CD configuré avec GitHub Actions ou Jenkins.

Risques et problèmes

Risque/Problème	Impact	Stratégie d'atténuation
Faux positifs/négatifs	Les tests peuvent donner des résultats incorrects	Revoir et améliorer continuellement
Couverture insuffisante	Certains modules peuvent ne pas être suffisamment testés	Utiliser des outils de couverture de c

Reporting et communication

- Fréquence des rapports : Rapports générés automatiquement après chaque exécution de test.
- Format des rapports : Rapports Jest (HTML, JSON) accessibles via le tableau de bord CI/CD.
- Parties prenantes : Équipe de développement, gestionnaires de projet, responsables métier.

Conclusion

Ce plan de test unitaire, utilisant le modèle A-A-A avec Jest, fournit une méthodologie rigoureuse pour assurer la qualité et la fiabilité de l'application mobile du restaurant. En suivant ce plan, l'équipe de développement pourra identifier et corriger les problèmes rapidement, garantissant une application robuste et prête pour le lancement.