Unit Test Plan

Module : User module, basket module, error-handling module , events module and promotions module	**Application :** Restaurant application
Tester: Ambassa Bienvenu, Meli Sonkoue Bernice, Ebene Dimene Annick Carole, Ngake Ngassam Brice and Tsapzeu Odilon Duval	

Exemple unit plan on User module

Module Overview

This module defines the the user's account details through validation and verification, and also it handles users account operations like purchases. This Modules contains 3 sub-modules (files) :

- account.js : It handles the user account creation and apply validation techniques. It also handles purchases by requesting a purchase history.

-user.js: It creates a user if this user does not exist in the system.

-purchaseHistory.js: it retrieves the purchase history from an external source.

Module Inputs

Users.js: userName (string): Email of the user userId (string):

Unique identifier for a user **Account.js:** eventName (string): Name

of the event tickets (number): Number of tickets purchased cost

(number): Cost of the purchase **purchaseHistory.js:**

purchaseData(object): includes event, ticket and cost.

userName (string): Email of the user

Module Outputs

Users.js:

Boolean: true if a user exist and false if not.

Id: Represents a new id created.

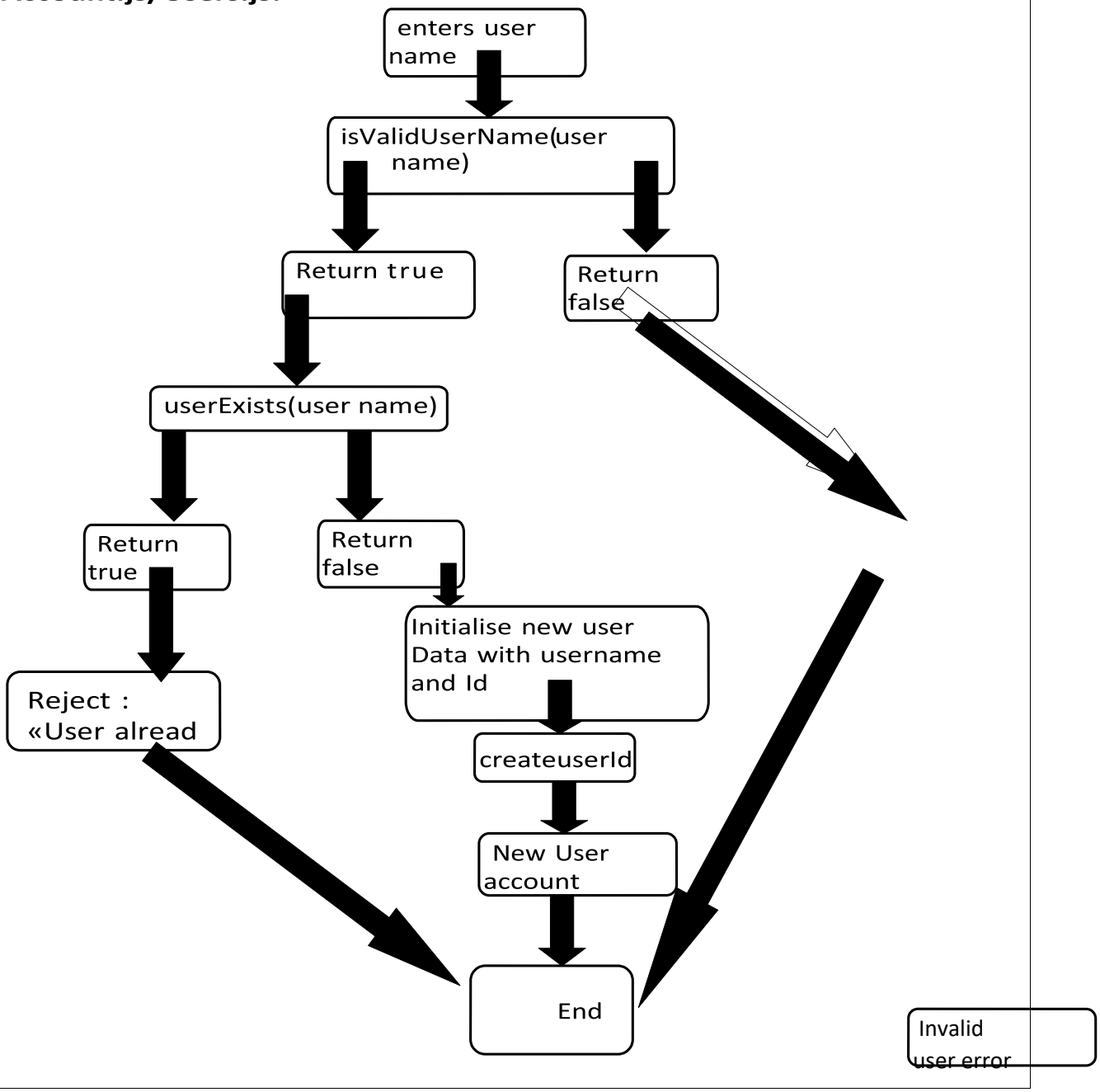
Account.js:

boolean: true if a username is valid and false if not. purchase History;
Represents the purchase history of the user. **purchaseHistory.js:**

request: An http request to get purchases of a user.

Logic Flow

Account.js/Users.js:



Test Data

Users.js

1) Test cases for User class

1.1) Should create User object with valid data

- new User(2, “pollah”)
- Expected outcome: Object created with properties {id: 2, username: “pollah”}
- Actual result: success (object created)

1.2) Should not create a user if invalid data tpes are passed

- new User(“lop”, 5)
- Expected outcome: error
- Actual result: Object created

2) Test cases for UserExists(username) function

2.1) Should return true if the user exist

- username: “newuser1@pluralsight.com”
- Expected outcome: true
- Actual result: true

2.2) Should return false if the user does not exist

- username: “newuser2@pluralsight.com”
- Expected outcome: false
- Actual result: false

3) Test cases for CreateUserId() functionuserExists(username)

3.1) Should return different user IDs on different calls

- createUserId() called 3 times
- Expected outcome: 3 different user IDs. Actual result: 1 ***account.js***

4) Test cases for Purchase class

4.1) Should create Purchase object with valid data

- new Purchase(“JUIN”, 4, 2400)
- Expected outcome: Object created with properties {eventName: “JUIN”, tickets: 4, cost: 2400} Actual result:

4.2) Should not create a Purchase object if ticket < 0

- new Purchase(“JUIN”, -3, 2400) Expected outcome: error Actual result:

4.3) Should not create a Purchase object if cost < 0

- new Purchase(“JUIN”, 4, -100)
- Expected outcome: error

- Actual result:

4.4) Should not create a Purchase object if cost is not an integer

- new Purchase("JUIN", 4, "bro")

Expected outcome: error Actual result:

4.5) Should not create a Purchase object if ticket is not an integer

- new Purchase("JUIN", "sap", 2400)

Expected outcome: error Actual result:

4.6) Should not create a Purchase object if eventName is not a string

- new Purchase(1, 4, 2400) Expected

outcome: error Actual result:

5) Test cases for isValidUserName(username) function

5.1) Should return true if the username is valid

- username: "newuser1@pluralsight.com" Expected outcome: true Actual result:

5.2) Should return false if the username is not valid

- username1: "newuser2-pluralsight.com"
- username2: "momo@monero-com"
- username3: "user.pamaro@polo"
- username4: 450
- username5: ' '
- username6: "user@.com"
- username7: "@domain.com" Expected outcome: false Actual result:

6) Test cases for createAccount(username) function

6.1) Should create an acount if the username is valid and user does not exist

- username: "newuser3@pluralsight.com"
- isValidUserName()=true
- userExists()=false
- Expected outcome: promise resolved with with data {userId: 2 , username:

"newuser3@pluralsight.com"} Actual result:

6.2) Should return an error message if the username is not valid

- username1: “newuser2-pluralsight.com”
- isValidUserName()=false
- Expected outcome: error message Actual result:

6.3) Should return an error message if the valid username exists alread

- username1: “newuser1@pluralsight.com”
- userExists()=true
- Expected outcome: promise rejected Actual result:

7) Test cases for getPastPurchases(username) function

7.1) Should return a purchase events if the userID is valid

- userID: 1
- Expected outcome: events Actual result:

7.2) Should throw an error if the userID is invalid

- userID: 67
- Expected outcome: error Actual result:

purchaseHistory.js

8) Test cases for getPurchasesHistor(userId) function

8.1) Should return a request if the userID is valid

- userID: 1
- Expected outcome: request Actual result:

8.2) Should return an error if the userID is invalid

- userID: “parole”
- Expected outcome: error Actual result:

8.3) Should return an error if the userID does not exist

- userID: 45
- Expected outcome: error Actual result:

9) Test cases for parsePurchasesResponse(purchaseData) function

9.1) Should return an arra of purchases if the purchaseData is valid

- purchaseData: {[“cobra”,3,40],[“dance”,6,75]}
- Expected outcome: arra of purchase

• Actual result: **9.2)** Should return an error if the purchseData is invalid • purchaseData1: 90 • purchaseData2:“cobra” • purchaseData: {[8,3],[“dance”,75]} □ Expected outcome: error □ Actual result:
Test Tools**: vitest**
TestCoverage: