TEST PLAN FOR THE E-COMMERCE STORE

COMP.SE.200-2022-2023-1

Software Testing

Petri Kreus (K421552)
Olli Sund (150162212)

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AUT	Application Under Test, E-Commerce Store
npm	Node Package Manager
nvm	Node version manager

1. INTRODUCTION

This document describes a test plan and a test design for the utility library of the front-end of the E-Commerce Store application. The aim is to give a detailed description of the test strategy, test objectives and test deliverables for the project.

First, descriptions of end-to-end scenarios are presented. Second, tools for the testing are defined with the actual tests. Finally, a Virtual Machine setup for the testing environment is illustrated.

2. DESIGN

In this section, we describe the general test design and its rationale. First, we define a scope for the test design. Second, the most important end-to-end scenarios of the application under test (AUT) are specified. Third, we present the main components that we identified based on the scenarios. Finally, a selection of source files to be tested are introduced.

2.1 Scope

The scope of the testing is to test 10 source files of the utility library, which consists of 43 source files in total. The amount to be tested is limited to 10 due to the time constraints. The test strategies are limited to unit test and integration tests, mostly since access to the application or any of its parts is not provided aside the utility library source files. That means that many testing strategies are left out of the scope, such as usability testing, system testing and acceptance testing.

Additionally, the specification states that only the top level of the utility library is to be tested, hence the files under the “`.internal/`” folder are left out of the testing scope. Due to the nature of the files under test, the testing concentrates only on functional tests.

To select the source files to be tested, first a few end-to-end scenarios are detected based on the provided specification of the product. Next, main components of the application are analysed from the scenarios. Finally, a prioritization method is used, utilizing the scenarios and components, to figure out the most important source files that need to be tested.

2.2 Scenarios

The testing is designed upon four main end-to-end scenarios identified from the application description. The scenarios described here are limited to four, since based on our evaluation, those cover the most important features of the application. Two of them are related to customer functionality and two to producer functionality. Following present these scenarios in a form of sequence diagrams.

Figure 1 describes user logging in, searching a product, adding a product to cart, and finally making a purchase. Figure 2 describes a new user registering an account and logging in. Figure 3 describes a producer logging in, adding a new product, and removing a product. Figure 4 describes a new producer registering a new account, waiting for it to be approved, and finally logging in after an approval.

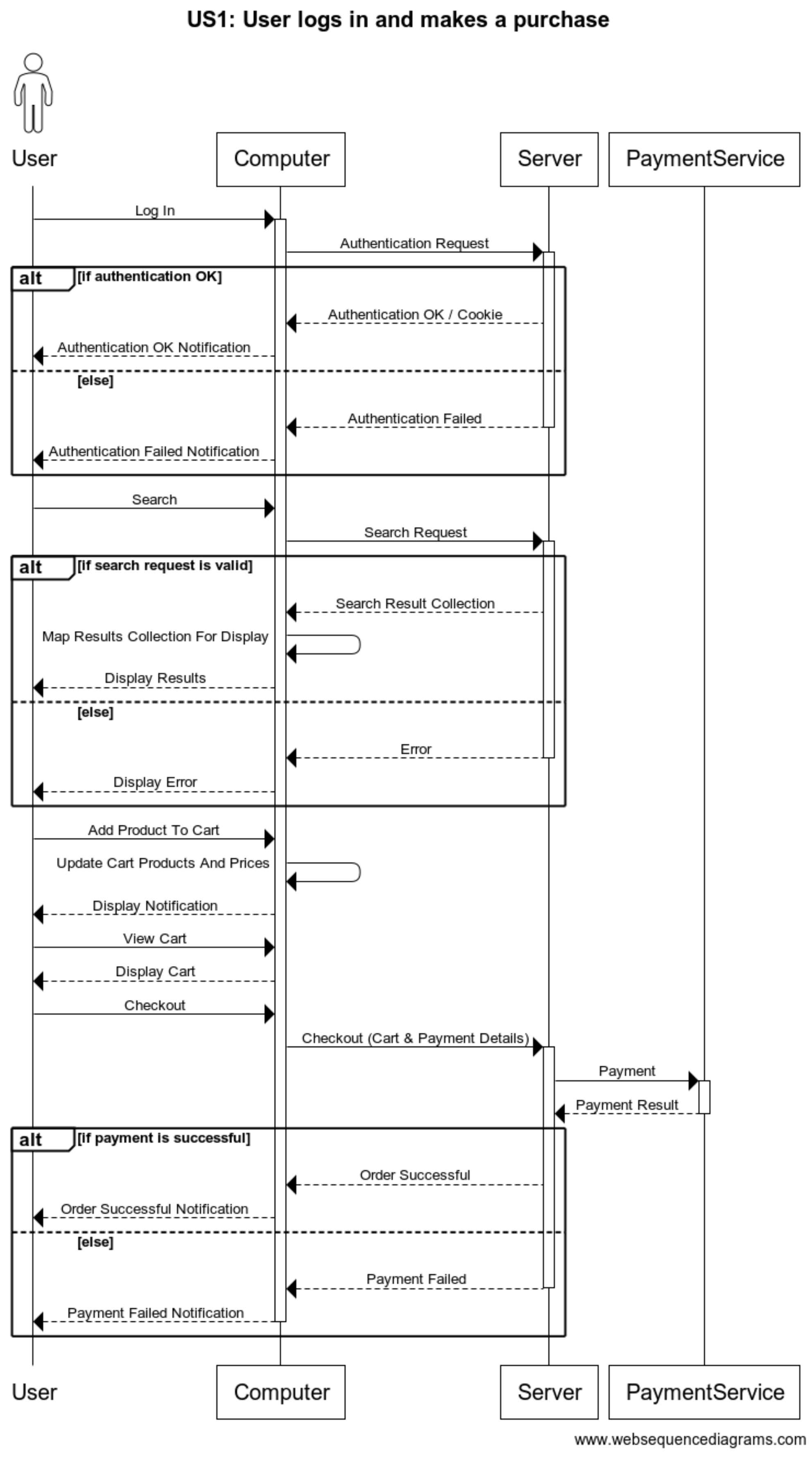

US1: User logs in and makes a purchase
User
Computer
Server
PaymentService
Log In
Authentication Request
alt [if authentication OK]
Authentication OK / Cookie
Authentication OK Notification
[else]
Authentication Failed
Authentication Failed Notification
Search
Search Request
alt [if search request is valid]
Search Result Collection
Map Results Collection For Display
Display Results
[else]
Error
Display Error
Add Product To Cart
Update Cart Products And Prices
Display Notification
View Cart
Display Cart
Checkout
Checkout (Cart & Payment Details)
Payment
Payment Result
alt [if payment is successful]
Order Successful
Order Successful Notification
[else]
Payment Failed
Payment Failed Notification
User
Computer
Server
PaymentService
www.websequencediagrams.com

Figure 1. *S1: User logs in and makes a purchase.*

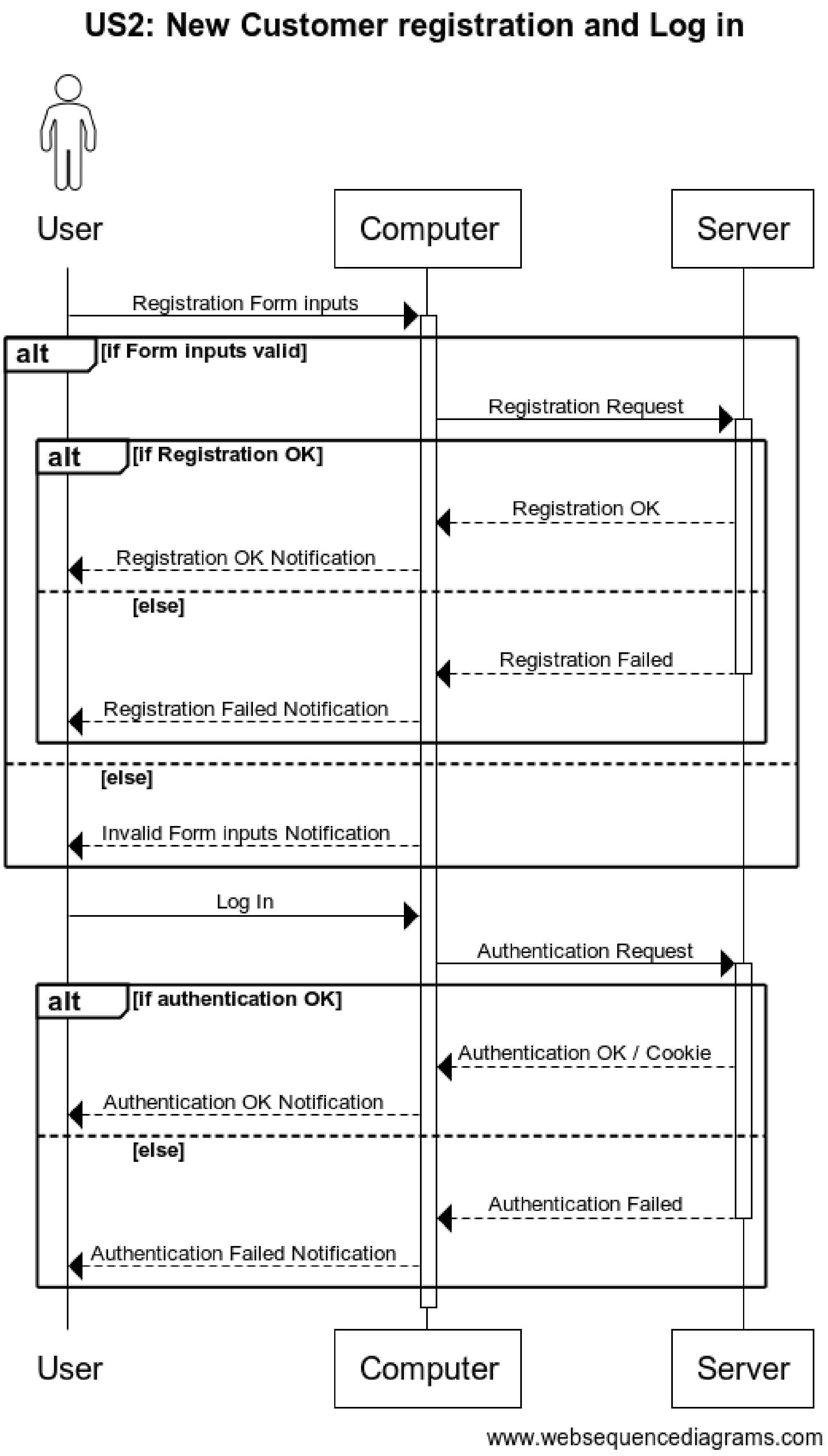

US2: New Customer registration and Log in
User
Computer
Server
Registration Form inputs
alt [if Form inputs valid]
Registration Request
alt [if Registration OK]
Registration OK
Registration OK Notification
[else]
Registration Failed
Registration Failed Notification
[else]
Invalid Form inputs Notification
Log In
Authentication Request
alt [if authentication OK]
Authentication OK / Cookie
Authentication OK Notification
[else]
Authentication Failed
Authentication Failed Notification
User
Computer
Server
www.websequencediagrams.com

Figure 2. *S2: New customer registers and logs in.*

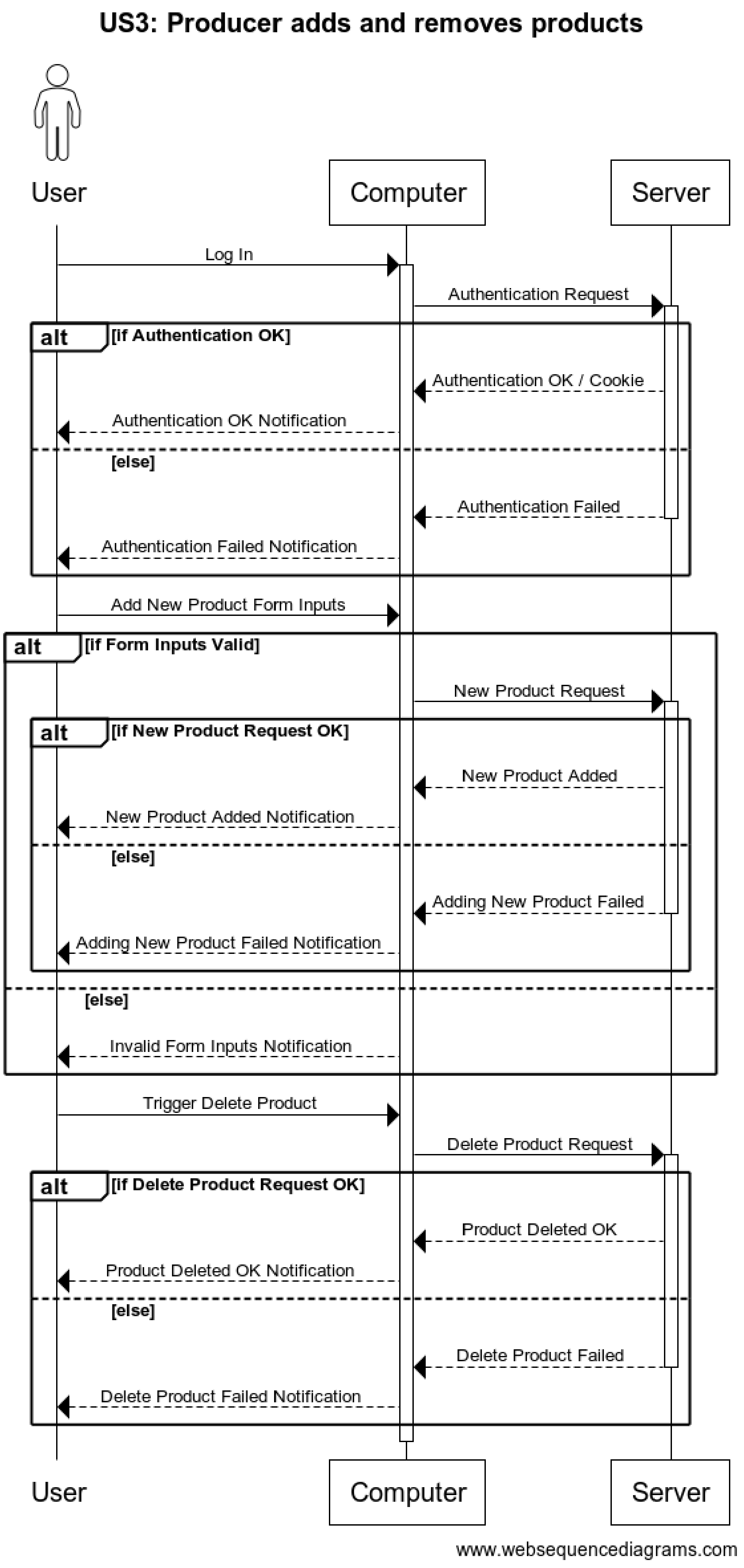

US3: Producer adds and removes products
User
Computer
Server
Log In
Authentication Request
alt [if Authentication OK]
Authentication OK / Cookie
Authentication OK Notification
[else]
Authentication Failed
Authentication Failed Notification
Add New Product Form Inputs
alt [if Form Inputs Valid]
New Product Request
alt [if New Product Request OK]
New Product Added
New Product Added Notification
[else]
Adding New Product Failed
Adding New Product Failed Notification
[else]
Invalid Form Inputs Notification
Trigger Delete Product
Delete Product Request
alt [if Delete Product Request OK]
Product Deleted OK
Product Deleted OK Notification
[else]
Delete Product Failed
Delete Product Failed Notification
User
Computer
Server
www.websequencediagrams.com

Figure 3. *S3: Producer adds and removes a product.*

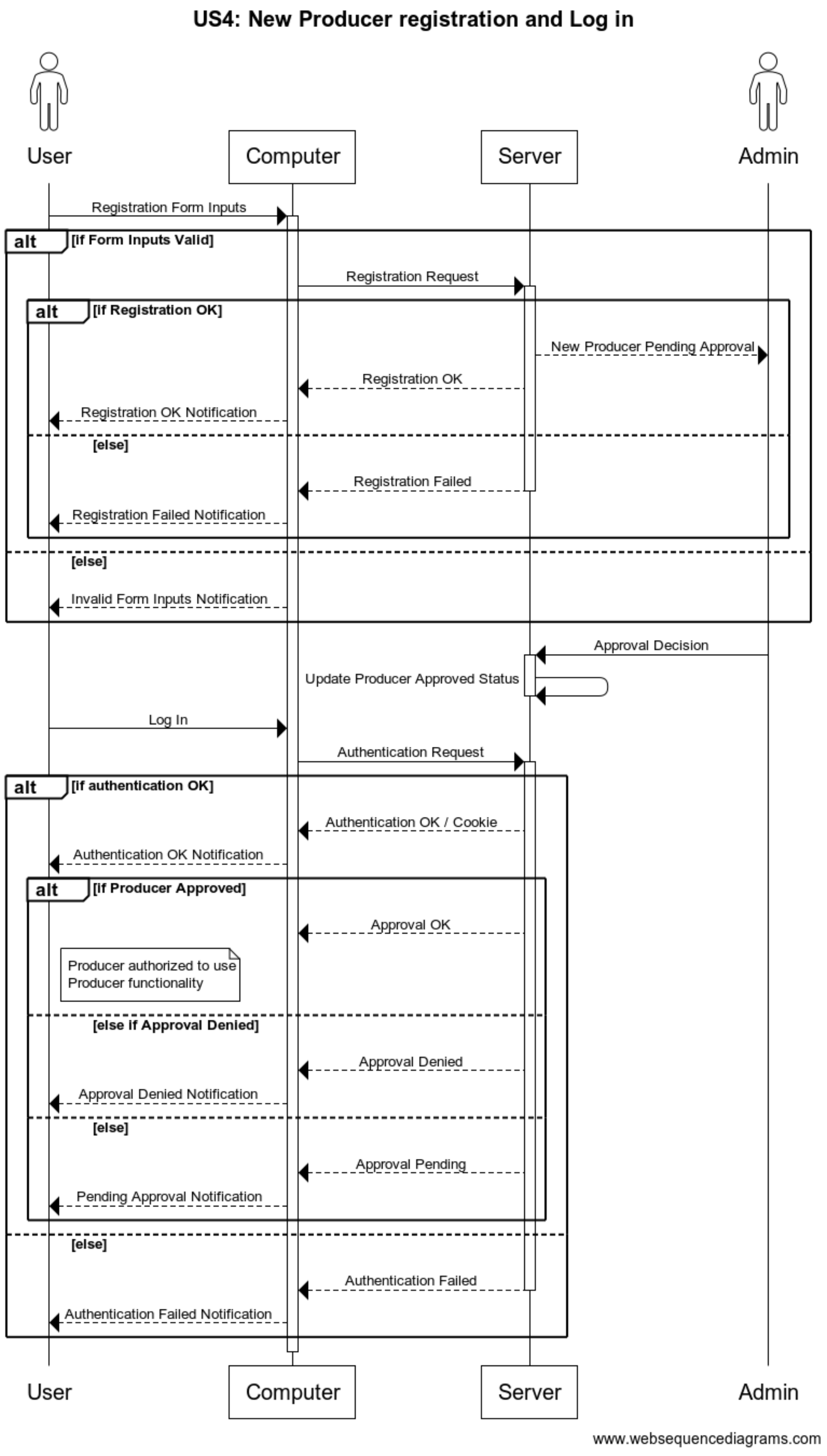

US4: New Producer registration and Log in
User
Computer
Server
Admin
Registration Form Inputs
alt [if Form Inputs Valid]
Registration Request
alt [if Registration OK]
New Producer Pending Approval
Registration OK
Registration OK Notification
[else]
Registration Failed
Registration Failed Notification
[else]
Invalid Form Inputs Notification
Approval Decision
Update Producer Approved Status
Log In
Authentication Request
alt [if authentication OK]
Authentication OK / Cookie
Authentication OK Notification
alt [if Producer Approved]
Approval OK
Producer authorized to use Producer functionality
[else if Approval Denied]
Approval Denied
Approval Denied Notification
[else]
Approval Pending
Pending Approval Notification
[else]
Authentication Failed
Authentication Failed Notification
User
Computer
Server
Admin
www.websequencediagrams.com

Figure 4. *S4: New producer registers and logs in.*

Based on these scenarios, we recognized five main functional components of the application. These components are:

- Log in / Registration (Customer & Producer)
- Product list (Customer & Producer)
- Shopping cart (Customer)
- Search (Customer)
- Product management (Producer).

2.3 Selected Source Files

With the scenarios and components identified, we first used the MoSCoW method to prioritize the source files. Due to the strict time limitations, the prioritization were conducted on a highly critical manner, and most of the files ended up under the Won’t test category from the start. In the end, we narrowed the source files down to the 10 Must test files, and all other files were moved under Won’t test category.

The source files included in the testing are displayed in the following table (Table 1).

Source File	Rationale for selection/Example use case	Related Scenario(s)
add.js	Basic math function, may be used in multiple situations.	S1, S3
at.js	May be used in many situations when getting data from an object.	S1, S2, S3, S4
defaultTo.js	May be used when displaying and saving information.	S1, S3
divide.js	Basic math function. May be used when displaying prices and in other calculations. Has a critical error in syntax.	S1, S3
filter.js	May be used when displaying product lists.	S1
isDate.js	May be used when displaying dates in product pages. Working with dates is a well-known challenge in JavaScript development.	S1, S3
isEmpty.js	May be used when checking validity of form fields.	S1, S2, S3, S4
map.js	May be used when converting array data into React components.	S1, S3
reduce.js	May be used when displaying prices in cart.	S1
words.js	May be used in parsing search parameters in product search.	S1

Table 1. *Selected source files with their selection rationale and relation to scenarios.*

3. TOOLS

To get the most out the testing, the production environment and the testing environment should be as similar as possible. However, the specification of the E-Commerce application does not describe the environment, aside from that its front-end uses React with the utility library provided. Thus, we will give our best guess, based on our research and experience, to describe a working environment for the basis of our tool selection. The environment description will concentrate only on the front-end React application and tools relating to the functional tests, which are specified in the next section.

3.1 General environment

As the front-end of the application is built with React, and the utility library has a "`package.json`" file, it will most likely run on Node.js [2]. Node.js is an open-source JavaScript runtime. It is a widely used base for various JavaScript applications. Node.js provides a useful package manager, Node Package Manager (npm), although other package managers could be used as well, such as pnpm or Yarn. With the npm, installing dependencies to the project is effortless, and controlling the versions of the dependencies is relatively easy.

The utility library provided will supposedly be installed to the production application with the npm, or similar.

3.2 Testing tools

We picked Jest [1], a JavaScript testing framework, for unit and integration testing. Since the testable file/function count is low, Jest will work well. It works without much additional configuration and is easy to use. If the testable amount was higher, a deeper consideration and comparison of various testing libraries should be conducted.

The Jest framework was tested by creating a simple test case for the divide function of the provided utility library.

If we had access to the React application, we suggest that React Testing Library [4] was to be used. It is a light-weight solution for testing React components. It provides utility functions with react-dom and react-dom/test-utils. React testing library is not a test runner or framework. Even though the library isn't specific to any framework it is recommended that React testing library would be used with Jest.

4. TESTS

In this section, the rationale behind the unit and integration tests cases is provided. Additionally, the test cases are specified in detail.

Due to the nature of the utility library, we posit that if the most important functionality is tested well with unit tests, the testing provides sufficiently clear picture of how the larger pieces, such as components, will work. That is why we concentrate on unit tests.

However, two integration tests are provided, basing on our best guess of how their related functionality could be implemented in the production application. It should be noted though, that complete component tests are nearly impossible without access to the application.

The application specification states that a Continuous Integration pipeline is used in the production. This should be considered in the design of the test cases.

The reports from the testing should include a test coverage report and a report of success state of the unit and integration tests.

Next, short descriptions of designed test cases for the selected functionality are presented. The tests cases are divided into unit tests and integration tests.

4.1 Unit Tests

Unit test cases are selected using a mix of Equivalence partition method and limit value analysis. The divide function should perform a simple mathematical division operation and the expected behaviour of the operation with given inputs should be quite straightforward. Unit tests are divided under the source files they are related to.

Test cases for divide.js

Divide.js is related to scenarios S1 and S3.

ID	TC001
Name	6 divided by 3 is 2
File	division.js
Type	Functional test, Positive test
Purpose	To test that the application display product's price per unit of sale correctly when user is searching for products (Scenario: US1)
Preconditions	division.js imported
Inputs	Positive dividend 6 and positive divisor 3
Expected Results	2
After-conditions	-

ID	TC002
Name	Dividing a positive number by zero returns Infinity
File	division.js
Type	Functional test, Negative test
Purpose	To test that the application handles illegal division by zero
Preconditions	division.js imported
Inputs	Positive dividend 10 and 0 as divisor
Expected Results	NaN
After-conditions	-

ID	TC003
Name	Dividing zero by zero returns NaN
File	division.js
Type	Functional test, Negative test
Purpose	To test that the application handles illegal division by zero
Preconditions	division.js imported, dividend and divisor variables defined
Inputs	Positive dividend 0 and 0 as divisor
Expected Results	NaN
After-conditions	-

ID	TC004
Name	Dividing -10 by 5 returns -2
File	division.js
Type	Functional test, Positive test
Purpose	To test that the function handles negative values correctly
Preconditions	division.js imported
Inputs	Positive dividend -10 and 5 divisor
Expected Results	-2
After-conditions	-

Test cases for filter.js

Filter.js is related to scenarios S1

ID	TC011
Name	Filter returns an array of objects where object property id equals 5
File	filter.js
Type	functional test, positive test
Purpose	To test that filter works as intended based on the examples given
Preconditions	Filter.js imported
Inputs	[{ id: 3 }, { id: 5 }, { id: 7 }, { id: 5 }]
Expected Results	[{ id: 5 }, { id: 5 }]
After-conditions	-

ID	TC012
Name	Filter returns an empty array with object.id === 9
File	filter.js
Type	functional test, positive test

Purpose	To test that the filter function works correctly with no matching re-sults
Preconditions	filter.js is imported
Inputs	[{ id: 3 }, { id: 5 }, { id: 7 }, { id: 5 }]
Expected Results	[] (empty array)
After-conditions	-

Test cases for words.js

Words.js is related to scenario S1

ID	TC021
Name	Input string is divided into individual search words
File	words.js
Type	functional test, positive test
Purpose	To test that the input string is correctly divided into words
Preconditions	Words.js is imported
Inputs	String: “one two three”, Delimiter pattern: ” “ (empty space)
Expected Results	[“one”, “two, “three”]
After-conditions	-

Test cases for map.js

Map.js is related to scenario S1

ID	TC031
Name	Square the numbers in array
File	map.js
Type	functional test, positive test
Purpose	To test that map function works correctly given simple function
Preconditions	Map.js is imported
Inputs	Array: [1, 2, 3, 4, 5], Function: (x) => x*x
Expected Results	[1, 4, 9, 16, 25]
After-conditions	-

Test cases for isEmpty.js

IsEmpty.js is related to scenarios S1, S2, S3 and S4.

ID	TC041
Name	Empty object {} returns true
File	isEmpty.js
Type	functional test, positive test
Purpose	To test that isEmpty works as specified with empty object

Preconditions	isEmpty.js is imported
Inputs	{} (empty object)
Expected Results	True
After-conditions	-

ID	TC042
Name	Empty array [] returns true
File	isEmpty.js
Type	functional test, positive test
Purpose	To test that isEmpty works as specified with empty array
Preconditions	isEmpty.js is imported
Inputs	[] (empty array)
Expected Results	True
After-conditions	-

ID	TC043
Name	Non-empty object returns false
File	isEmpty.js
Type	functional test, positive test
Purpose	To test that isEmpty works as specified with non-empty object
Preconditions	isEmpty imported
Inputs	{ productId: 1, description: “Eggs” }
Expected Results	False
After-conditions	-

ID	TC044
Name	Non-empty array returns false
File	isEmpty.js
Type	functional test, positive test
Purpose	To test that isEmpty works as specified with non-empty array
Preconditions	isEmpty.js is imported
Inputs	[1, 2, 3]
Expected Results	false
After-conditions	-

Test cases for add.js

Add.js is related with scenario S1 and S3.

Note: String addition is not tested because there is no use case for it.

ID	TC051
Name	5 plus 5 equals 10
File	add.js
Type	functional test, positive test
Purpose	To test that add function works as specified with positive values
Preconditions	Add.js is imported
Inputs	Addends 5 and 5
Expected Results	10
After-conditions	-

ID	TC052
Name	5 + -5 is 0
File	add.js
Type	functional test, positive test
Purpose	To test that add function works as specified when mixing positive and negative values
Preconditions	Add.js is imported
Inputs	Addends 5 and -5
Expected Results	0
After-conditions	-

Test cases for at.js

At.js is related to scenarios S1, S2, S3 and S4.

ID	TC061
Name	Returns an array with values matching given paths
File	at.js
Type	functional test, positive test
Purpose	To test that at function works as specified with nested object and ar-ray of paths
Preconditions	At.js is imported
Inputs	Object: { ‘a’: [{ ‘b’: { ‘c’: 3 } }, 4] }, Array of paths: [‘a[0].b.c’, ‘a[1]’]
Expected Results	[3, 4]
After-conditions	-

ID	TC062
Name	Returns an array with undefined values when called with integers
File	at.js
Type	functional test, negative test
Purpose	To test that at function works as specified with unexpected input values
Preconditions	At.js is imported
Inputs	1, 2
Expected Results	[undefined]
After-conditions	-

Test cases for defaultTo.js

DefaultTo.js is related to scenarios S1 and S3.

ID	TC071
Name	Returns a default value when value is null
File	defaultTo.js
Type	functional test, positive test
Purpose	To test that defaultTo works as specified with null input value
Preconditions	defaultTo.js is imported
Inputs	value: null, default value: “I’m a default value :)”
Expected Results	“I’m a default value :)”
After-conditions	-

ID	TC072
Name	Returns a default value when value is undefined
File	defaultTo.js
Type	functional test, positive test
Purpose	To test that defaultTo works as specified with undefined input value
Preconditions	defaultTo.js is imported
Inputs	value: undefined, default value: “I’m a default value :)”
Expected Results	“I’m a default value :)”
After-conditions	-

ID	TC073
Name	Returns the value when a valid value is given
File	defaultTo.js
Type	functional test, positive test
Purpose	To test that defaultTo works as specified with valid input value
Preconditions	defaultTo.js is imported
Inputs	value: “I’m valuable XD”, default value: “I’m a default value :)”
Expected Results	“I’m valuable XD”
After-conditions	-

Test cases for reduce.js

Reduce.js is related to scenarios S1.

ID	TC081
Name	Calculate the sum of array elements
File	reduce.js
Type	functional test, positive test
Purpose	To test that reduce function works as specified with valid values
Preconditions	Reduce.js is imported
Inputs	Collection: [1, 2, 3], iteratee: (sum, n) => sum + n), initial accumulator: 0.
Expected Results	6
After-conditions	-

Test cases for isDate.js

IsDate.js is related to scenarios S1 and S3.

ID	TC091
Name	Returns true when argument is a valid Date object
File	isDate.js
Type	functional test, positive test
Purpose	To test that isDate works as specified with valid input
Preconditions	isDate.js is imported
Inputs	new Date()
Expected Results	True
After-conditions	-

ID	TC092
Name	Returns false when argument is not a valid Date object
File	isDate.js
Type	functional test, positive test

Purpose	To test that isDate function works as specified with invalid input
Preconditions	isDate.js is imported
Inputs	‘Mon April 23 2012’
Expected Results	False
After-conditions	-

4.2 Integration Tests

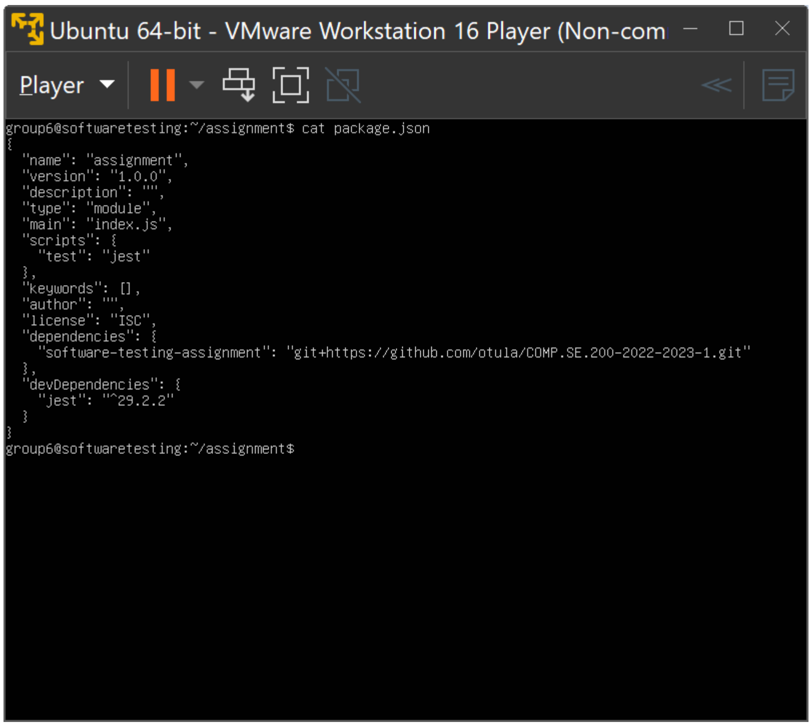
ID	TC101
Name	Calculate total price from an object array
Files	add.js, at.js, reduce.js
Type	functional test, positive test
Purpose	Test a part of possible cart component. (Scenario US1)
Preconditions	add.js, at.js, and reduce.js are imported
Inputs	An array of objects with price property: [{ price: 5 }, { price: 10 }, { price: 15.90 }]
Expected Results	30.90
After-conditions	-

ID	TC102
Name	Filter a product array with a string of multiple words to return an array of correct products
Files	filter.js, words.js
Type	functional test, positive test
Purpose	Test a part of possible search component; Help to alleviate ambiguity: Should the search results be inclusive or exclusive related to the search words?
Preconditions	filter.js and words.js are imported; An array of objects with productName property defined: [{ productName: “Fresh eggs” }, { productName: “The best lettuce ever” }, { productName: “Very fresh potatoes“ }]
Inputs	“fresh potato”
Expected Results	[{ productName: “Fresh eggs” }, { productName: “Very fresh potatoes” }]
After-conditions	The array of objects deleted

VIRTUAL MACHINE

In this section, a virtual machine (VM) setup is provided. A common environment helps with multiple possible issues, such as the common “it works on my machine” problem.

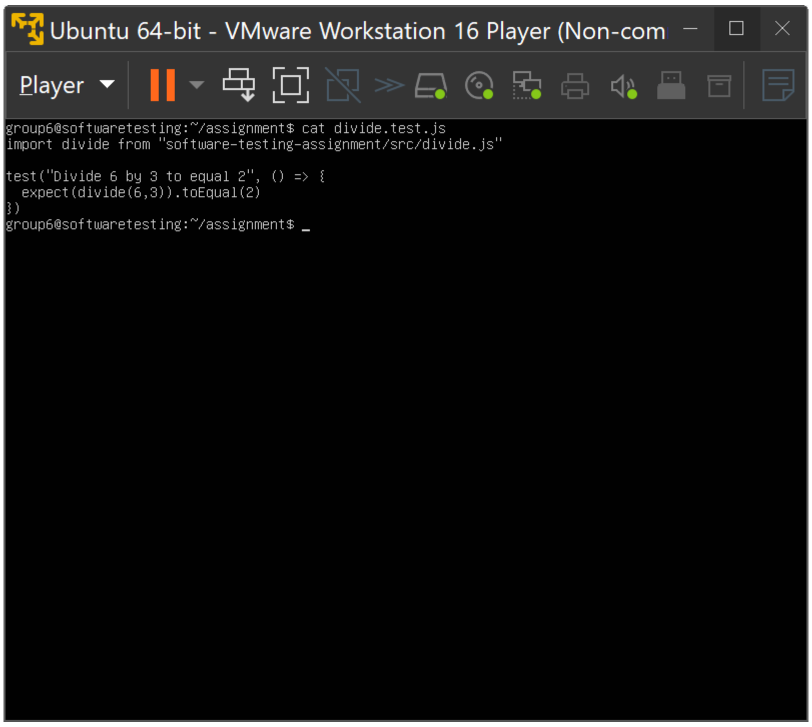
We use a VMware Workstation Player [5] to implement a VM to be used for the testing. The operating system chosen is Linux based Ubuntu Server. On Ubuntu, a Node version manager, nvm, is installed, so that we can easily install and run the tests on multiple versions on Node [3]. Additionally, we install the Jest testing framework as a development dependency and the provided utility library as a global dependency with the npm package manager. The contents of the package.json file is provided in the Figure 5.



```
group6@softwaretesting:~/assignment$ cat package.json
{
  "name": "assignment",
  "version": "1.0.0",
  "description": "",
  "type": "module",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "software-testing-assignment": "git+https://github.com/otula/COMP.SE.200-2022-2023-1.git"
  },
  "devDependencies": {
    "jest": "^29.2.2"
  }
}
group6@softwaretesting:~/assignment$
```

Figure 5. *Contents of the package.json on the VM*

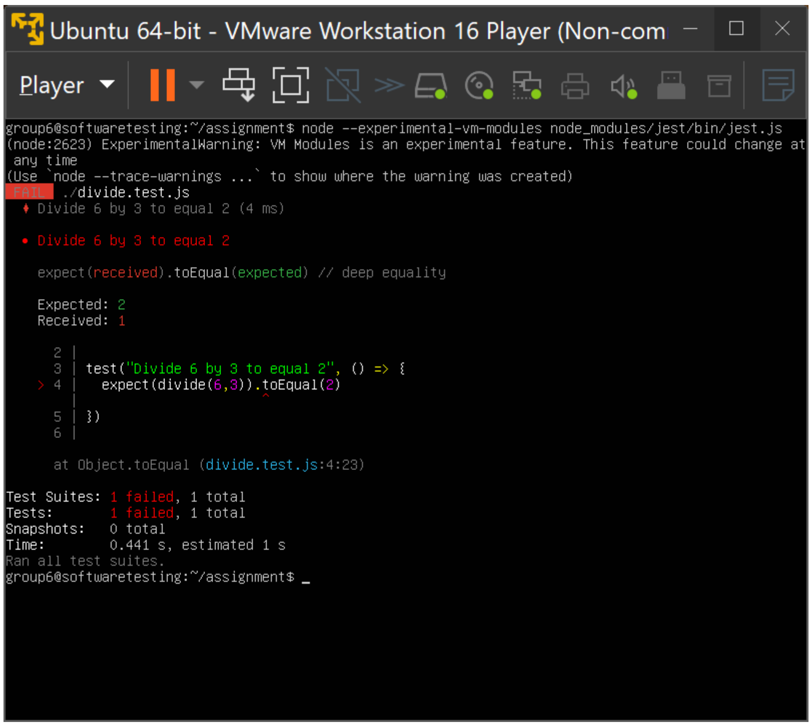
To test the VM implementation, a simple Jest test for the divide.js source file of utility library was created and ran. The test file, divide.test.js, is displayed in Figure 6 and the test results in the Figure 7.



```
group6@softwaretesting:~/assignment$ cat divide.test.js
import divide from "software-testing-assignment/src/divide.js"

test("Divide 6 by 3 to equal 2", () => {
  expect(divide(6,3)).toEqual(2)
})
group6@softwaretesting:~/assignment$ _
```

Figure 6. *Contents of the divide.test.js file on the VM.*



```
group6@softwaretesting:~/assignment$ node --experimental-vm-modules node_modules/jest/bin/jest.js
(node:2623) ExperimentalWarning: VM Modules is an experimental feature. This feature could change at
 any time
(Use `node --trace-warnings ...` to show where the warning was created)
 FAIL  ./divide.test.js
  ♦ Divide 6 by 3 to equal 2 (4 ms)

  • Divide 6 by 3 to equal 2

    expect(received).toEqual(expected) // deep equality

    Expected: 2
    Received: 1

      2 |
      3 | test("Divide 6 by 3 to equal 2", () => {
    > 4 |   expect(divide(6,3)).toEqual(2)
        |                       ^
      5 | })
      6 |

      at Object.toEqual (divide.test.js:4:23)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.441 s, estimated 1 s
Ran all test suites.
group6@softwaretesting:~/assignment$ _
```

Figure 7. *Test results of a simple test, provided by Jest.*

REFERENCES

[1] Jest. (2022). https://jestjs.io/

[2] Node.js. (2022). https://nodejs.org

[3] nvm. (2022). https://github.com/nvm-sh/nvm

[4] Testing Library. (2022). https://testing-library.com/docs/react-testing-library/intro/

[5] VMware. (2022). https://www.vmware.com/products/workstation-player.html