E-COMMERCE APPLICATION TEST REPORT

COMP.SE.200 SOFTWARE TESTING - TUNI

GitHub: https://github.com/heidicompse200/software_testing.git
Coveralls: https://coveralls.io/github/heidicompse200/software_testing

Heidi Seppi, H252889

Definitions, acronyms and abbrevations

AI	Artificial intelligence
E-commerce	Electronic commerce
CI	Continuous integration

Introduction

This E-commerce store test report focuses on describing the tests what have been implemented and reporting the found bugs from the different test runs. The purpose of this document is to explain how E-commerce store was tested and what was found during those tests so that the found bugs can be fixed. If more testing is found needed later on for E-commerce store, all the future testing can be documented on this same document so all the information of the tests can be found from the same place.

First, the testing framework and CI pipeline are explained and GitHub Actions workflow is shown. All the alterations what have been made to the test plan [Appendix B] are also explained here. Next, the found bugs from the done tests are reported. This chapter also focuses on estimating the overall quality of the tested libraries and if they would be ready for production. In addition, the overall state of the E-commerce store is being analyzed here. In the next chapter any use of AI during testing is described and overall advantages and disadvantages of AI usage are being discussed. Lastly, some feedback will be given about the overall experience of the course.

Tests & CI pipeline

The main idea was to create tests for the E-commerce store to test out if the provided functions work as intended and if they have any bugs in them. These test have been published in our GitHub [1] repository. GitHub actions [2] were used to create CI pipeline for the tests and to link Coveralls [3] to the repository. We had issues with Coverall reports and after so many tries, we weren't able to get Coveralls working with GitHub Actions. So we weren't able to get any Coveralls reports from our test runs. Coveralls would have provided important information regarding the test coverage and other statistics so it is a shame that we couldn't get it to work with our GitHub project.

GitHub Actions was used to create CI pipeline for running the implemented tests automatically. Basic idea is that tests are always run on push and pull commands when they have been done on main branch. Npm install is run first to install all the required packages using package.json. After that, npm test script runs all of implemented tests. Below is our action workflow file which was used to create the pipeline.

Workflow:

```
name: Node.js CI
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [18.x, 20.x, 22.x]
        # See supported Node.js release schedule at https://nodejs.org/en/about/releases/

    steps:
    - uses: actions/checkout@v4
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v4
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'
    - run: npm install
    - run: npm test

    - name: Coveralls
      uses: coverallsapp/github-action@v2
```

The tests for the E-commerce store application were implemented using JavaScript test framework Mocha [4] with the help of Chai assertion library [5]. With these tools the unit

tests were created for the selected 10 files which were documented on the test plan [Appendix B]. E-commerce store had lots of more files in addition to these 10 selected ones but due to time-constraints and limited resources, these 10 files were seen as the most crucial ones to be tested for the E-commerce store. So, unit test were made for files: chunk.js, slice.js, toInteger.js, words.js, filter.js, reduce.js, get.js, defaultTo.js, eq.js and for isEmpty.js.

The types of tests we created are unit tests and integration tests. Integration tests were mainly created for the function chunk.js since it also uses slice.js and toInteger.js so the tests done for the chunk.js also test if all of these functions work together as intended. isEmpty.js tests also test if the isEmpty function works with isBuffer.js fuction. toInteger.js function uses toInfinite.js for infinite values so also the integration of these functions is being tested during toInteger.js tests.

These tests can be run locally with the following instructions:
Git and and Node.js with npm is required to be able to run the tests locally.
1. Clone the GitHub repository https://github.com/heidicompse200/software_testing.git to your local machine.
2. Navigate to software_testing folder.
3. Run *npm install* to install needed files.
4. Run *npm test* to run all the test.

In total there are 75 tests created for the selected files. Most of the tests try to first test the most basic use of the function and after that try to cover different kind of parameter options and boundary values which could cause errors on the code. Below you can find test log of the run tests in which total of 70 test passed and 5 failed. These found bugs have been documented on the bug report which is introduced in the next chapter. In the test log, the tests have been divided so that each file's test is under the same section. The test name describes what the test is about. In addition, failures have been highlighted with red and passed tests highlighted with green colour.

Test log:

Testing the Chunk Function

1) Testing basic functionality of chunk with array=[a,b,c,d] and length=2

✓ Testing the situation when chunk length is zero (0)

✓ Testing if function throws error when chunk is longer than the array

2) Testing when chunk cant be split evenly. Final chunk should be the remaining elements

Testing the defaultTo Function

✓ Testing basic functionality of defaultTo with value=1 and defaultValue=20

✓ Testing defaultTo with value=undefined and defaultValue= ´default ´

✓ Testing defaultTo with value=null and defaultValue= ´default ´

3) Testing defaultTo with value=NaN and defaultValue= ´default ´

✓ Testing defaultTo with value=null and defaultValue=undefined

Testing the eq Function

✓ Testing basic functionality of eq by comparing two same numbers

✓ Testing eq by comparing two same strings

✓ Testing eq by comparing two similar strings but with different lengths

4) Testing eq by comparing two different types (string and number) with similar looking values

✓ Testing eq by comparing two different types (string and array) with different looking values

✓ Testing eq by comparing two different objects: {a:1} and {a:2}

✓ Testing eq by comparing two different objects which are almost the same but other one has more elements

✓ Testing eq by comparing NaN and NaN

✓ Testing eq by comparing two same numbers which are opposite signs (+ and -)

✓ Testing eq by comparing +0 and -0

✓ Testing eq by comparing undefined and undefined

✓ Testing eq by comparing null and null

✓ Testing eq by comparing two boolean values: true and true

✓ Testing eq by comparing two boolean values: true and false

✓ Testing eq by comparing two boolean values: false and false

Testing the Filter Function

✓ Testing basic functionality of filter function with object of users and filtering them by active field

✓ Testing filter function with array of numbers from which only >2 number should be returned to their own array

✓ Testing filter function with an empty array

✓ Testing filter function when no elements match the filtering predicate

Testing the Get Function

✓ Testing get function when trying to the first level element from an object with string path format

✓ Testing get function when trying to the second level element from an object with string path format

✓ Testing get function when trying to the first level element from an object with array path format

✓ Testing get function when trying to the second level element from an object with array path format

✓ Testing get function when the element trying to be accessed is not found from the object

✓ Testing get function when object is null or undefined

✓ Testing get function when defaultValue for undefined results is 0 instead of undefined (default option)

Testing the isEmpty Function

✓ Testing basic functionality of isEmpty with parameter 1 (type number)

✓ Testing isEmpty when Map size is bigger than zero (has content)

✓ Testing isEmpty when Map size is 0

✓ Testing isEmpty when Set size is bigger than zero (has content)

✓ Testing isEmpty when Set size is 0

✓ Testing isEmpty when Object is enumarable

✓ Testing isEmpty when Object isnt enumarable

✓ Testing isEmpty when string value is ´´ (length is zero)

✓ Testing isEmpty when string value is "test" (length bigger than zero)

✓ Testing isEmpty when string values has length of 1 (for example ´t´)

✓ Testing isEmpty when boolean value is true

5) Testing isEmpty when boolean value is false

✓ Testing isEmpty when the value is array which has no elements

✓ Testing isEmpty when the value is array which has elements

✓ Testing isEmpty when the value is null

✓ Testing isEmpty when the value is undefined

Testing the Reduce Function

✓ Testing basic functionality of reduce by calling it: reduce([1, 2], (x, y) => x * y, 1))

Testing the Slice Function

✓ Testing basic functionality of slice with array=[1, 2, 3, 4] without start and end index

✓ Testing basic functionality of slice with array=[1, 2, 3, 4] and start slicing index=2

✓ Testing basic functionality of slice with array=[1, 2, 3, 4], start index=1 and end index=3

✓ Testing slicing an array with zero (0)

✓ Testing slicing an empty array

✓ Testing slicing when slicing start index is bigger than the array length

✓ Testing slicing when slicing end index is bigger than the array length

✓ Testing slicing with negative start slicing index

✓ Testing slicing with negative end slicing index

✓ Testing slicing with start index=1, end index=5 and with an array=[a,b,c,d,e,f,g]

✓ Testing slicing when end is before start index. Expecting empty array as a result

✓ Testing slicing when start and end are the same indexes. Expecting empty array as a result

Testing the toInteger Function

✓ Testing changing Float number to Integer

✓ Testing changing negative Float number to Integer

✓ Testing changing Float number to Integer which would in this case need to be rounded down using number = 1.5.

✓ Testing changing Infinite number to Integer. It is supposed to be 1.7976931348623157e+308.

✓ Testing changing MIN_ VALUE to Integer. It is supposed to be 0.

✓ Testing changing String value of number to Integer with the test string="1"

✓ Testing changing string="testi" to Integer. This is supposed to return 0 and not supposed to throw error.

Testing the Words Function

✓ Testing basic functionality of words with string= ´one, two, three & four´ without pattern

✓ Testing basic functionality of words with string= ´one, two, three & four´ with pattern=/[^,]+/g

✓ Testing words function when string is empty

✓ Testing words function when string contains special characters: äöå!?

70 passing (27ms)

5 failing

1) Testing the Chunk Function

Testing basic functionality of chunk with array=[a,b,c,d] and length=2:

AssertionError: expected [['c', 'd'], undefined] to deeply equal [['a', 'b'], ['c', 'd']]

+ expected - actual

```
[
  [
+   "a"
+   "b"
+ ]
+ [
    "c"
    "d"
  ]
]
```

at Context.<anonymous> (file:///home/runner/work/software_testing/software_testing/test/chunk.test.js:9:46 at process.processImmediate (node:internal/timers:476:21)

2) Testing the Chunk Function

Testing when chunk cant be split evenly. Final chunk should be the remaining elements:

AssertionError: expected [[4, undefined, undefined], ...(1)] to deeply equal [[1, 2, 3], [4]]

+ expected - actual

```
[
  [
+   1
+   2
+   3
+ ]
+ [
    4
-   [undefined]
-   [undefined]
  ]
]
```

at Context.<anonymous> (file:///home/runner/work/software_testing/software_testing/test/chunk.test.js:35:46) at process.processImmediate (node:internal/timers:476:21)

3) Testing the defaultTo Function

Testing defaultTo with value=NaN and defaultValue=´default´:

AssertionError: expected NaN to equal 'default'

at Context.<anonymous> (file:///home/runner/work/software_testing/software_testing/test/defaultTo.test.js:19:45) at process.processImmediate (node:internal/timers:476:21)

4) Testing the eq Function

Testing eq by comparing two different types (string and number) with similar looking values:

AssertionError: expected true to equal false

+ expected - actual

-true

+false

at Context.<anonymous> (file:///home/runner/work/software_testing/software_testing/test/eq.test.js:19:30) at process.processImmediate (node:internal/timers:476:21)

5) Testing the isEmpty Function

Testing isEmpty when boolean value is false:

AssertionError: expected true to equal false

+ expected - actual

-true

+false

at Context.<anonymous> (file:///C:/Users/heidi/Koulu/software_testing/software_testing/test/isEmpty.test.js:63:35) at process.processImmediate (node:internal/timers:478:21)

Changes done to the part 1 test plan:

Original test plan can be found from Appendix B. We modified the way we wanted report the found bugs. We created a new bug reporting template which can be found from Appendix A and also the word template is attached to the assignment 2 zip file. This template was used to write all the necessary information regarding the found bugs. We didn't use originally planned test documentation template because we got the test log with test descriptions printed out after running the tests so this template wasn't needed to report the done tests.

Findings and conclusions

While running the implemented tests, all the found bugs were reported using bug report template [appendix A]. Below you can find the bug report in which the found bugs have been described.

Bug report from done tests

Title: Chunk function doesn’t produce correct output when slicing the array

Test Date: 2024-12-08

Tester: Heidi Seppi

Status: Open

Test Type: Integration testing

Test Environment: Test framework Mocha with Chai

Tested files: chunk.js, slice.js

Purpose of the test: Functionality of chunk.js and slice.js was being tested. Chunk.js uses functionality of slice.js so the idea of the integration tests was to assure that these functionalities work together, and the output is correctly splitted array.

Description of the bug: Chunk function doesn’t produce correctly sliced array as an output. Output of function call chunk([‘a’,’b’,’c’,’d’], 2) was [['c', 'd'], undefined] and the correct output would had been [[‘a’,’b’],[’c’,’d’]].

Bug type: Functional bug

Possible to reproduce: Yes

How to reproduce: Call chunk function from the chunk.js with parameters: array=[‘a’,’b’,’c’,’d’] and chunk length=2 and check what the function returns. Output of chunk([‘a’,’b’,’c’,’d’], 2) is supposed to be [[‘a’,’b’],[’c’,’d’]].

Possible reason for the bug: Programming error

Severity: 4

Likelihood of occurrence: 4

Comments: -

Title: Chunk function doesn’t produce correct output when chunk can’t be split evenly

Test Date: 2024-12-08

Tester: Heidi Seppi

Status: Open

Test Type: Integration testing

Test Environment: Test framework Mocha with Chai

Tested files: chunk.js, slice.js

Purpose of the test: Functionality of chunk.js and slice.js was being tested. Chunk.js uses functionality of slice.js so the idea of the integration tests was to assure that these functionalities work together, and the output is correctly split array.

Description of the bug: Chunk function doesn't produce correctly sliced array as an output when chunk can't be split evenly. It should give for the a second array element the rest of the original array but now instead it gives the rest of the array and 'undefined'. Output of chunk([1,2,3,4], 3) was [[4, undefined, undefined], [1, 2, 3]] and correct expected output would had been [[1, 2, 3], [4]].

Bug type: Functional bug

Possible to reproduce: Yes

How to reproduce: Call chunk function from the chunk.js with parameters: array= [1,2,3,4] and chunk length=3 and check what the function returns. Correct output of chunk([1,2,3,4], 3) is [[1, 2, 3], [4]].

Possible reason for the bug: Programming error

Severity: 4

Likelihood of occurrence: 4

Comments: -

Title: defaultTo function doesn't return default value when value is NaN

Test Date: 2024-12-09

Tester: Heidi Seppi

Status: Open

Test Type: Unit testing

Test Environment: Test framework Mocha with Chai

Tested files: defaultTo.js

Purpose of the test: defaultTo function was being tested to ensure that it returns the correct default value for different inputs.

Description of the bug: When calling the defaultTo function with parameters value=NaN and defaultValue='default', function should return the defaultValue. Instead it returns NaN.

Bug type: Functional bug

Possible to reproduce: Yes

How to reproduce: Call the defaultTo function from defaultTo.js file with parameters value=NaN and defaultValue='default' (defaultTo(NaN, 'default') and see what the function returns. Function should return inserted defaultValue ('default').

Possible reason for the bug: Programming error

Severity: 4

Likelihood of occurrence: 3

Comments: Error can most likely be found from defaultTo.js file.

Title: eq function doesn't return false when comparing different types with similar looking values

Test Date: 2024-12-09

Tester: Heidi Seppi

Status: Open

Test Type: Unit testing

Test Environment: Test framework Mocha with Chai

Tested files: eq.js

Purpose of the test: eq function was being tested to ensure that it returns the correct bool type of comparison results when comparing different kind of values.

Description of the bug: When calling the eq function to compare different type of variables, eq function should return false. Now, when string and number type of variables which had similar looking values were compared, the eq function returned falsely true.

Bug type: Functional bug

Possible to reproduce: Yes

How to reproduce: Call the eq function from eq.js file with number and string type of parameters 1 and '1'. Check if the function returns true or false. Eq function should return false when two different types are being compared.

Possible reason for the bug: Programming error

Severity: 3

Likelihood of occurrence: 4

Comments: Error can most likely be found from eq.js file.

Title: isEmpty function doesn't return false when using Boolean value of false as a parameter

Test Date: 2024-12-09

Tester: Heidi Seppi

Status: Open

Test Type: Unit testing

Test Environment: Test framework Mocha with Chai

Tested files: isEmpty.js

Purpose of the test: isEmpty function was being tested to ensure that it returns the correct bool type of result when checking if different types are empty.

Description of the bug: When calling the isEmpty function with parameter false, it returned true as an ouput. In the isEmpty.js file it was described that 'Objects are considered empty if they have no own enumerable string keyed' but the description didn't mention how true/false values should be handled. In the example cases isEmpty(true) returns true so on this logic it would be assumed that isEmpty(false) returns false.

Bug type: Functional bug

Possible to reproduce: Yes

How to reproduce: Call the isEmpty function from isEmpty.js file with parameter false (Boolean). Check if the function returns true or false. Check if in the documentation is has been described if Boolean false should return false or true.

Possible reason for the bug: Programming error or documentation error

Severity: 3

Likelihood of occurrence: 3

Comments: Should be investigated if the wanted return value for parameter false is Boolean true or false.

Below you can find a table about the tested files in which quality of testing has been analyzed. If the test coverage information would had been possible to get from Coveralls, then even more detailed test quality analysis could had been done.

File name	Test quality and coverage
chunk.js	Two bugs were found from chunk.js library so these need to be fixed before chunk.js should be taken into use. After the fixes, tests for chunk.js should be run again to ensure that no new bugs have been accidently created. The library didn't inform if undefined values are accepted on the result array, or if the array should be empty instead of having some undefined values. For this it would be good to check from the customer, what kind of output can be accepted and modify the library according to that. In addition, more unit tests for different types of inputs could be implemented to ensure better test quality and coverage.
slice.js	Slice.js has lots of different unit tests which use different slicing indexes and cover possible error cases. All the unit tests are passed and tests do cover variety of different inputs so according to this, slice.js seems to be ready to put in use.
toInteger.js	toInteger.js tests try to cover boundary values of different inputs (such as infinite values) and all the implemented tests are passed so according to this information, toInteger.js library could be taken into use.
words.js	These tests focus on testing out different string inputs with and without a pattern. We have only 4 unit tests implemented so more tests are probably needed to ensure better test coverage. At least more test cases about different kinds of inputs should still be implemented.
filter.js	Filter unit test try to cover different kind of input scenarios and possible error cases but we have only 4 tests implemented so to ensure better test coverage, more test with different kinds of filter function and predicates should be implemented. So far, all the tests are passed.
reduce.js	reduce.js has only one unit test implemented so test coverage isn't good for this library. The unit test only test out one basic functionality of reduce function so more unit tests about different kinds of inputs and outputs are necessary before the reduce.js library can be put in use.
get.js	get library has lots of unit tests implemented which try to test different kinds of inputs with different objects and path formats and some possible error cases. All the tests are passed so get.js could be put in use.
defaultTo.js	Unit tests focus on different types of inputs (such as NaN and undefined values) and seem to cover the most crucial cases. One of the unit tests

	fails so this needs to be fixed and after the fix, tests should be run again. If no more tests fail, defaultTo.js library can be used.
eq.js	eq library unit test cover variety of different inputs focusing on boundary values. One of the tests don't pass so there needs to be a fix for this. After running the tests again after the bug fix to ensure that all the tests still pass, this library is ready to be used.
isEmpty.js	isEmpty library tests focus on different types of inputs (such as number, integer and string values) and try to cover so that all the different types of data will be handled the correct way. Also, Nan, undefine etc. value cases are being tested out. One of the tests doesn't pass so this should be first investigated. After fixing the bug and running tests again so ensure no more bugs can be found, I would put this library in use.

Overall, the libraries are not ready to put in production since there was bugs found from some of them and some libraries would need better variety of test cases to ensure that they work as intended with different kind of data. So, we wouldn't recommend putting the E-commerce application into production yet with these libraries. Unit tests and integration tests were provided for the most crucial libraries due to time-constraint reasons these test don't cover fully all the libraries what are used in E-commerce store. So at least system testing should be implemented to ensure that the libraries work together as a whole and all the listed end-to-end scenarios in the test plan work as intended with the libraries. User interface testing is also needed for the user interface of E-commerce store. After that, possible acceptance testing should be done to ensure that customer is satisfied with the application before putting the application into production.

AI and testing

AI was not used when implementing the tests and when writing the test report. Mainly we wanted to focus on creating the tests ourselves so that we were able to think and learn ourselves what kind of test situations we should cover and how to find different boundary values to test out. If we would had used AI to create the tests, then we wouldn't have learned to think and analyze different testing possibilities. AI can be also useful because after implementing the tests yourself, you could use it find more scenarios to test which you haven't thought about before. So, it can be used as an efficient tool to make more comprehensive tests. For this assignment, we didn't have enough time to use AI to find more test scenarios unfortunately.

If AI is used too much especially when trying to learn about testing, then the learning isn't as efficient as it would be when you would try to sort out the tests yourself. Also, one major thing about AI is that it can hallucinate so the tests written by AI should always be checked and analyzed if they are useful for the certain case.

Course feedback

I think the assignments were fun to do and helped me to learn how to plan out the testing and how to actually implement the tests yourself. It was also useful to analyze yourself if the implemented tests are enough for the E-commerce store application or what kind of tests would still be needed so that the application can be put into production with the provided libraries. I liked the idea of being able to plan and implement tests yourself.

I learned a lot about different types of testing (such as integration and system testing) and especially about how detailed bug reports need to be so that the bugs can be fixed efficiently. I had done unit tests before with Mocha using Chai library but this course gave me better understanding of how to find different boundary values to test and how the different inputs can be divided into their equivalent classes for which the tests can be implemented. Especially I found it interesting to learn more during one of the course practical exercises how to test user interfaces (like web pages). I found it difficult to link Coveralls and in the end, I couldn't make the Coveralls work with GitHub Actions. GitHub Actions were also a new thing for so at first I found it quite difficult to do the workflow for the assignment but after some time, I got better understanding of it.

In general, course's practical exercises were useful and interesting and I would have liked to have practical exercise sessions every week because I felt like I learned the most during those sessions. I liked the assignment as well but sometimes I felt like more instructions especially about how to install and use the tools would have been nice to have.

References

[1] GitHub (https://github.com/)

[2] GitHub Actions (https://github.com/features/actions)

[3] Coveralls (https://www.npmjs.com/package/coveralls)

[4] Mocha (https://www.npmjs.com/package/mocha)

[5] Chai (https://www.npmjs.com/package/chai)

Appendix

Appendix A: Bug report template

Bug report from done tests

The purpose of this document is to be able to gather all the found bugs to the same template.

Title: *Title for the bug report*

Test Date: *Date when the test had been done in a form YYYY-MM-DD*

Tester: *Tester's name*

Status: *Status of the reported bug: open/investigating/closed*

Test Type: *Type of testing, for example unit testing or system testing*

Test Environment: *Test framework what was used*

Tested files: *Files which were under test*

Purpose of the test: *Why the test was done and why it was needed*

Description of the bug: *Describe in detail what kind of bug was found*

Bug type: *What kind of bug it is, for example syntax error or logic error*

Possible to reproduce: *Yes/No*

How to reproduce: *Explain in detail how to reproduce the bug and leave this field empty if it's not possible.*

Possible reason for the bug: *What could cause the bug, for example programming error or design error*

Severity: *Scale 1-5 (1=cosmetic, 5=clients lose money or are harmed)*

Likelihood of occurrence: *Scale 1-5 (1=unusual or incorrect use, 5=every time the system is used)*

Comments: *Any comments or extra information what is needed to know.*

Appendix B: Test plan (assignment part 1)

Test plan has also been attached to the assignment 2 zip file.

E-COMMERCE APPLICATION TEST PLAN

COMP.SE.200 SOFTWARE TESTING - TUNI

Tatu Kankare, 151182440
Heidi Seppi, H252889

Definitions, acronyms and abbrevations

AI	Artificial intelligence
E-commerce	Electronic commerce
UML	Unified modeling language
Sequence diagram	Interaction diagram showing how and when the needed actions of the process are happening

Introduction

The purpose of this document is to plan out how an E-commerce application defined in the course assignment will be tested. Main idea of this E-commerce application is to sell food products from different producers. Users can browse products using different search criteria, add products to the shopping cart and checkout when they are ready to purchase the selected products. On the other hand, producers can add their own products to the application for users to purchase.

First, the most common end-to-end scenarios are described and visualized using sequence diagrams and after that the selected tools for testing are described. After the scenario and testing tool chapters, the document focuses on describing the planned tests and lists the 10 chosen source files from the utility library for which the unit tests will be implemented later. Lastly, the way to report and document tests will be discussed about. Also, the use and benefits of AI tools are mentioned at the end of this documentation.

Scenarios

End-to-end scenarios define application workflows from the beginning to the end. We have defined four most common end-to-end scenarios for the described E-commerce store. These end-to-end scenarios are described below with the help of UML sequence diagrams. The assignment description didn't mention any use of databases but for this case it is assumed that at least the products are stored in a database.

Scenario 1: User browses and searches for different products by category, price, product content and producer information.

This scenario describes the situation when the user wants to browse the products and search for certain products from the E-commerce store. The user can browse different products using category, price, product content and producer information to find suitable products for them.

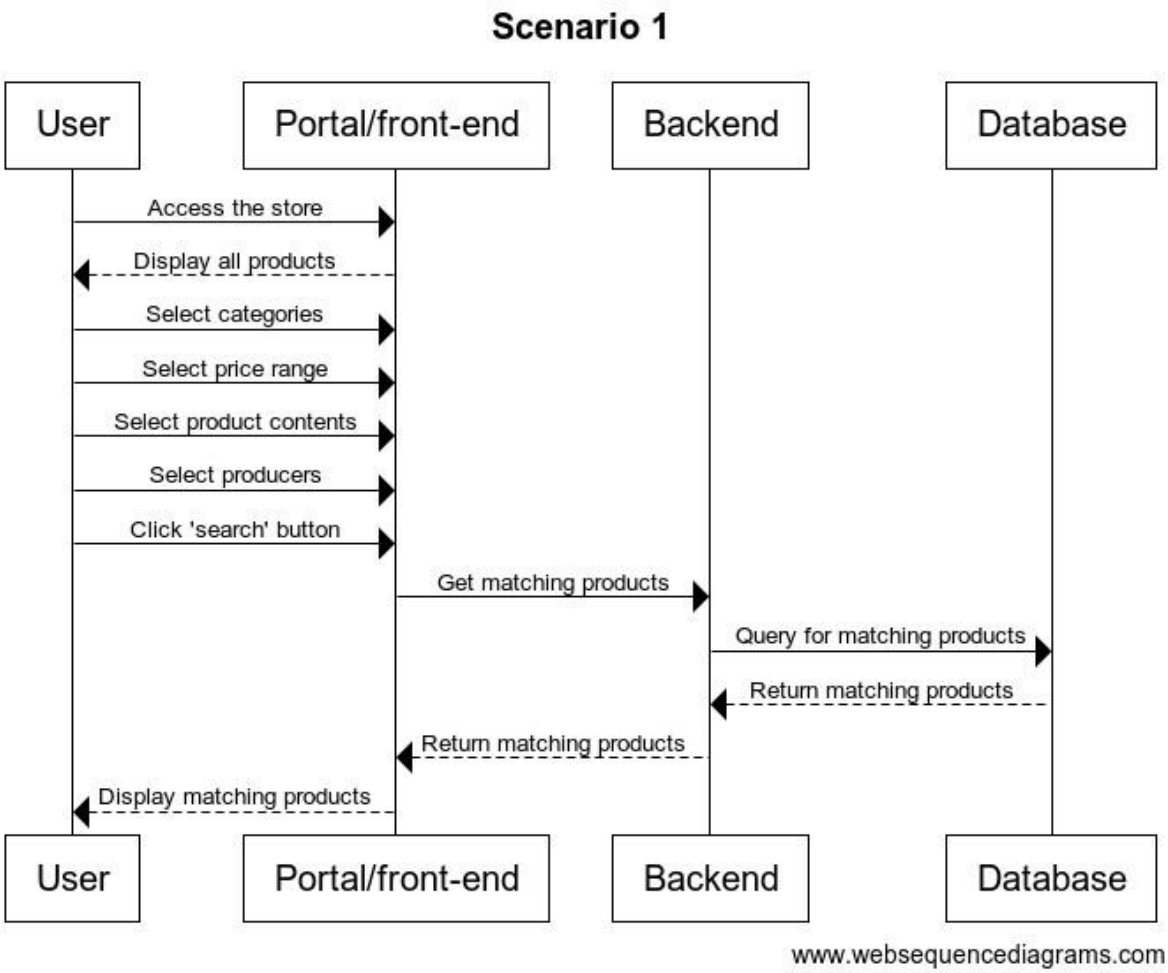

Figure 1: Sequence diagram for the scenario when user is browsing different products on the store.

Figure 1 sequence diagram shows the main actions when user wants to browse for different E-commerce store products. The user can at first see all the products and if they want to, search for certain kinds of products by selecting category, price range, product content and/or producer. Products are stored in the database so the backend will make a query to find matching products and return the results to the frontend. After the search the store displays the products to the user which match the selection criteria. The user can make new searches with different search criteria.

Scenario 2: User adds or removes products from the shopping cart and views the total price.

This scenario is about the user adding selected products to their shopping cart and being able to see the total price of their shopping cart. The user is also able to delete products from their shopping cart and the total price is updated automatically. In the E-commerce store assignment description, there were no direct mentions of being able to delete products from the shopping cart, but this functionality was considered here because removing items from the shopping cart is usually one of the basic

functionalities in online shops. Figure 2 displays the sequence diagram for this scenario.

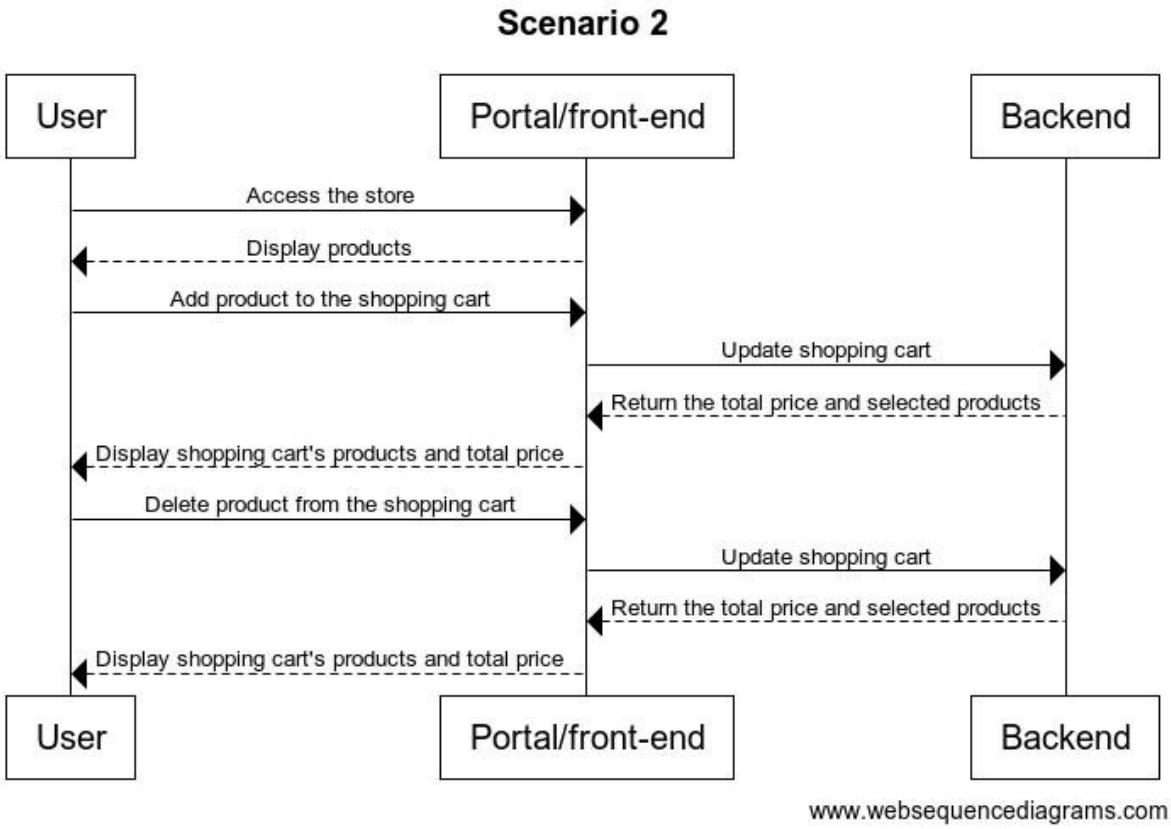

Figure 2: Scenario 2 of adding and removing products from the shopping cart.

Scenario 3: User purchases the products which have been added to the shopping cart.

Scenario 3 describes a situation where the user adds products to the shopping cart and wants to buy the selected products. Figure 3 shows the sequence diagram for this scenario.

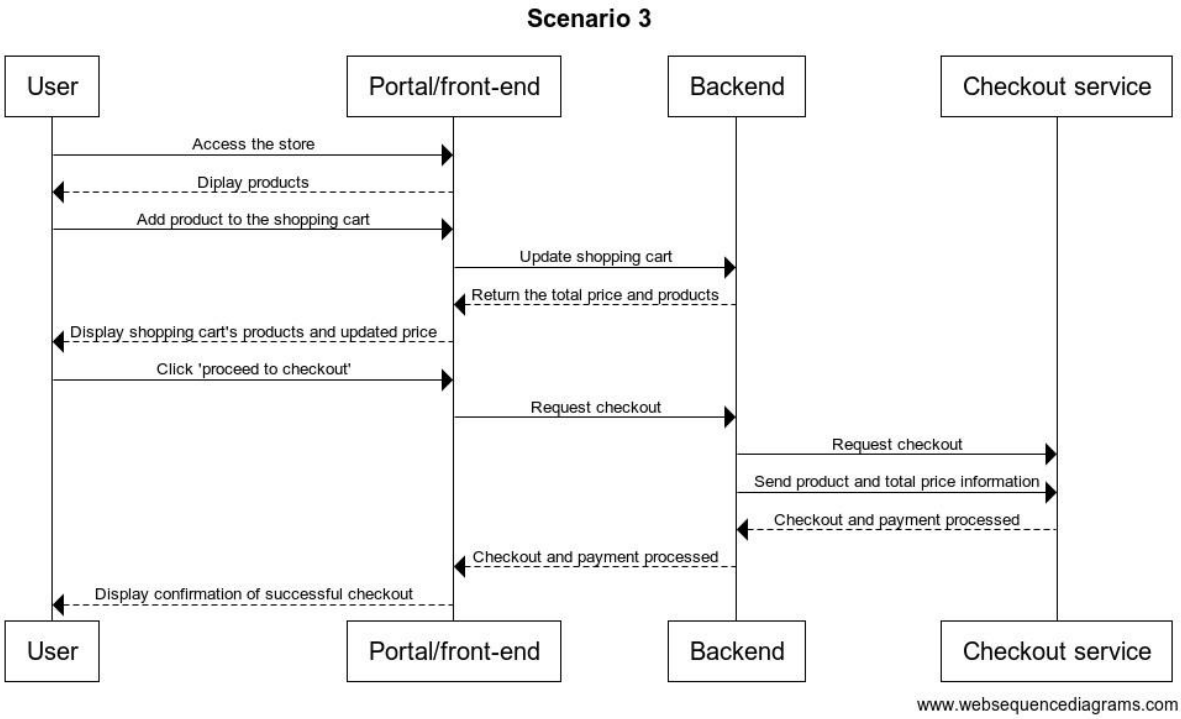

Figure 3: Scenario 3 of user doing checkout after adding products to the shopping cart.

At first user adds the needed products to the shopping cart and when they are ready to purchase the products, they can click ‘proceed to checkout’ button to proceed with the ordering. This begins the checkout process, and the third-party checkout service handles the checkout and payment methods. After the payment has been accepted and the ordering has been completed in the third-party service, the user gets confirmation of successful checkout. In this scenario it is assumed that the checkout service handles the ordering and the payment of the products completely.

Scenario 4: A food producer adds products for sale using the portal or front-end application.

A food producer needs to be able to add their products to the store and the scenario 4 shown in figure 4 describes this situation. In this case all the products are assumed to be saved to a database in which the new added product will be saved.

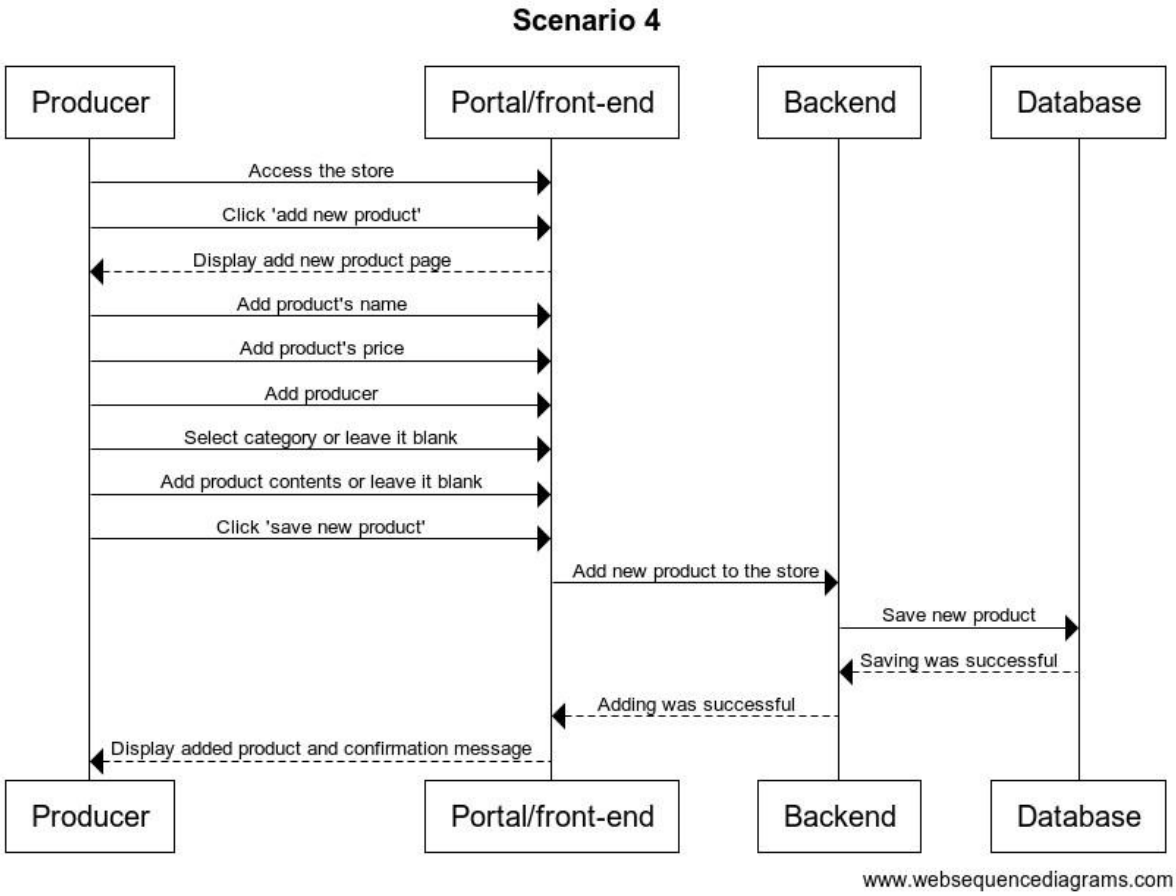

Figure 4: Scenario 4 of producer adding a new product to the store.

Producer can start adding a new product by clicking the button 'add new product' which then opens a page where all the product information for the new product can be filled. Product needs to have at least a name, a price and a producer. Category and product content is optional to fill in. After all the required information has been filled, producer can save the product. Product will be saved to the product database, and the producer will get a confirmation message when the product has been added successfully to the site.

Tools

While making the test plan and visualizing the end-to-end scenarios, WebSequenceDiagrams [1] drawing tool was used to create sequence diagrams. After testing this tool out, it seemed to be the most efficient way to draw the diagrams for the different scenarios.

The assignment definition requires the unit testing environment to include Github [2], Github Actions [3] and Coveralls [4].

Mocha unit testing framework [5] will be used for the unit tests. Chai assertion library [6] may also be used. They have been found to be an effective way of testing JavaScript in other projects.

Tests

Unit tests will be performed on the library [7] files specified in table 1 using appropriate test automation methods. Limited integration testing may be performed, where the selected components work together in a manner making integration possible.

Table 1: Files of the Library to be Unit tested.

Priority	File	Rationale for testing
High	chunk.js	Displaying products in a grid.
High	slice.js	Dependency of chunk.js
High	toInteger.js	Dependency of chunk.js
High	words.js	Search matching
High	filter.js	Search filtering
High	reduce.js	Calculating total price
Medium	get.js	Generic utility
Medium	defaultTo.js	Generic utility
Medium	eq.js	Generic utility
Medium	isEmpty.js	Generic utility

Other functions in the library were considered low priority due to their high specificity of use and lack of obvious use cases and have been excluded from unit testing due to time constraints (only 10 source files can be tested). Unit tests for the selected files try to find errors from the main actions mentioned in the end-to-end scenarios. For example, creating unit tests for chunk.js is important so that it is possible to find errors happening while displaying the store products.

System testing will be performed manually: testing through the described end-to-end scenarios 1,2,3 and 4. Errors from both users and third-party services will be included in some of the test runs. System testing ensures that the recognized end-to-end scenarios work as a whole and the different E-commerce parts, such as front-end,

back-end and third-party services, work together successfully so that the core requirements for the E-commerce store are being met.

Usability testing of the front-end interfaces both for users and producers will be performed to ensure they meet accessibility guidelines and can be operated without a difficulty if a person is unfamiliar with this particular web store.

Extensive performance testing will not be necessary. Due to the small scope of the store and serving small local businesses, large amounts of users are not expected. Sufficient overprovisioning of resources should minimize the possibility of significant performance issues not found during other testing.

If the E-commerce store application hasn't been released to the customer and is not yet in use, some form of customer acceptance testing should be performed to make sure that the customer requirements have been understood in a correct way. Acceptance testing helps to make sure that the customer is satisfied with the final application.

Content left out of testing

The given E-commerce store assignment description didn't mention any login and registration system for users so this won't be thoroughly tested since the documentation didn't describe clearly if the application has this feature and what kind of login and registration system would be needed. Although some form of access control system will likely be required for the producers to prevent misuse. In this case it will be assumed to be provided as a third-party service, but its interface may be tested.

The checkout service is specifically provided as a third-party service and as such will not be tested directly, however the interface of the checkout service may be tested.

The assignment description didn't mention if any databases are being used so it is assumed that at least the products and their information is stored in some database. Tests won't directly focus on database usage, but system testing makes sure that all the end-to-end scenarios work as intended.

Test reporting and classification

Test results for tests other than unit tests will be documented according to a template in Appendix A. Unit test results may also be documented if the defects found by them are not fixed immediately.

Found defects will be categorized based on their likelihood of occurrence (1-5, 1=unusual or incorrect use, 5=every time the system is used) and severity (1-5, 1=cosmetic, 5=clients lose money or are harmed).

Tests will be considered passed once no more defects are found during the specified test cases. Adequate test coverage is dependent on the quality of these tests.

AI and testing

It is possible to use AI, for example ChatGPT [8], to find different end-to-end scenarios for the E-commerce store application using the given store description. AI might give new ideas of what the E-commerce store could do and how the different features could work but it is not recommended to blindly trust all the information what AI gives since it might not always suit the exact situation perfectly. AI could also help to find new areas to be tested from the application which weren't taken into consideration originally. For example, in this case AI could give reasons why performance testing should be considered to be more important than it was originally considered to be. Although it is good to note that AI might hallucinate so it can give false information so it should only work as a tool to help to give new ways to think how the testing could be done.

During this assignment AI was not used because the main idea of the assignment is to learn yourself how to plan the testing for the E-commerce store application. One of the best ways to learn is to think and consider different solutions yourself and not just trust AI to give you direct answers how the testing should be done. Once the person has more experience of testing applications, AI can help by giving new different ideas of how to test components.

References

[1] WebSequenceDiagrams (https://www.websequencediagrams.com/)

[2] GitHub (https://github.com/)

[3] GitHub Actions (https://github.com/features/actions)

[4] Coveralls (https://www.npmjs.com/package/coveralls)

[5] Mocha (https://www.npmjs.com/package/mocha)

[6] Chai (https://www.npmjs.com/package/chai)

[7] Assignment Library (https://github.com/otula/COMP.SE.200-2024-2025-1)

[8] ChatGPT (https://chatgpt.com/)

Appendix

Appendix A: test_documentation_template.docx

Template can be found attached to the submission zip file. A snip image of the test documentation template is shown below.

Test Date: *YYYY-MM-DD*

Tester: (*If applicable)*

Test Type: *System, performance etc.*

Test Environment: *System, browser, test framework etc. (If applicable)*

Test Description: *What was tested, how it was tested*

Defects found: *Description of found defects and how to reproduce them*

Defect	How to reproduce	Severity	Likelihood of occurrence