Testing plan

COMP.SE.200-2022-2023-1 Software Testing
Roosa Kuusivaara 284599
Kalle Nyman 284272

Table of contents

Definitions, acronyms and abbrevations

SUT: System under test, the software that being is tested.

CI-pipeline: Continuous Integration pipeline. A script that runs tests on any modifications to the source code.

BDD: Behaviour-driven development. In short, creating software in a way that reflects user behaviour.

Introduction

The document contains a test plan for a web application, E-commerce store. The application is an online store, that sells food from various small producers. The user can search products by category, price, product contents, add them to a shopping cart and buy them. Food producers can add their products to the online store via portal or by using a front-end application.

The purpose of the document is to have a clear test plan for the application. In the document we surveyed what is important to test and what is not so important, how we are going to test the components, what tools we are using and what kinds of tests should be done.

In the document we introduce four different end-to-end **scenarios** for the common use cases of the website. We go through the components and chosen tools what will be used for the testing. There is a UML diagram of one of our scenarios and there is a list of all the source code files we selected for unit tests.

After Scenarios -chapter, we have the **Tools** -chapter which includes a broader description of the tools we have chosen and an explanation of why we selected them and how the tools would be used. The last part of the document is about the **Tests**, how the central functionality of the website needs to be tested, bug and issue categorization and code coverage.

Scenarios

Scenarios

Following is a description of end-to-end scenarios that a user of the website might go through.

Scenario 1 Basic shopping

-> Search for product -> add 3 products to cart

-> remove one from cart -> go to checkout

-> Checkout -> go back to search (*customer forgot one product)*

-> add product to cart

-> checkout -> payment -> buy products

Scenario 2 Producer adds new product (portal)

Food producer portal -> log in / register -> add product

Scenario 3 Producer adds new product (front- end app)

Food producer front end app -> log in / register ->add product

Scenario 4 Empty cart

-> add product to cart

-> remove it

-> go to checkout -> pay (with empty cart, should obviously not be allowed)

Main functionalities of the store that are tested are listed in the Tests-chapter.

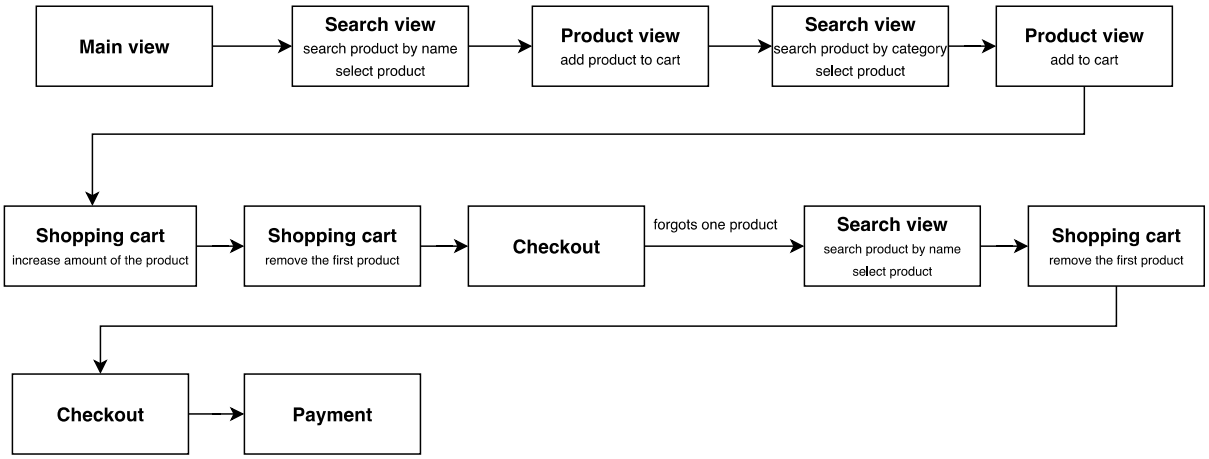
Chosen tools

We will use *Mocha* test framework for writing tests for the utility library. Utilities: https://github.com/otula/COMP.SE.200-2022-2023-1

- Test framework: *Mocha* https://mochajs.org
- reporter: *Mochawsome* https://github.com/adamgruber/mochawesome
- assertion library: *Chai* https://www.chaijs.com
- code coverage: *Coveralls* https://coveralls.io
- Issue tracker: *Github issue tracker*
- CI-pipeline: *GitHub Actions*

UML diagram

A diagram of our first scenario.



Chosen utility library functions (files) for testing

add

capitalize

compact

countBy

eq

filter

isArrayLike

toString

toNumber

memoize

Tools

Descriptions of the chosen tools:

Mocha

Mocha is an open-source unit test framework for JavaScript that runs on nodeJS. It is perhaps the most popular framework for unit testing in JavaScript. The purpose of Mocha for testing the application will be the creation and management of test cases and suites. When paired with the other tools it can be used to form a comprehensive testing for the application. We chose Mocha because we are already familiar with it, and because it is easy to use. An example test suite and case from Mocha website:

```
describe('#indexOf()', function () {
    it('should return -1 when the value is not present', function () {
      assert.equal([1, 2, 3].indexOf(4), -1);
    });
  });
```

“Describe“ constructs a test suite and “it” contructs a test case. We tested Mocha and Chai in a previous course WebDevelopment 1 and on this course in the first practical exercise.

Chai

Chai is a BDD assertion library that can be used in mocha tests. As with Mocha, we chose it because of familiarity and ease of use. The assertions are written in a very plain-english type way, almost anyone can understand the meaning of the assertions very easily. For example, with a testing variable “foo” we could perform a check:

```
foo.should.be.a('string');
```

to affirm that the variable type is String.

Coveralls

Using Coveralls for code coverage is mandatory according to the project description, so we chose it because of that. Code coverage is important for knowing how much of the source code gets executed during a testing run.

Github issue tracker and Github Actions

Github issue tracker and Github Actions makes sense because they are free and because the utility library is hosted there. Issue tracker is used for reporting bugs that are revealed for example through testing. Actions is to create a CI-pipeline which runs the tests every time modifications are pushed to the utility library. So, the importance for testing the application is that this is the way the unit tests are run, at least when pushing to version control. We have used these tools at work and on the Web Development 1 course.

Tests

The scenarios in the *Scenarios*-chapter should be validated with system level GUI tests, performed by a human tester. Unit tests and integration tests are also needed. As the online store consists of multiple subsystems (payment & checkout, producer portal) there must be system integration tests for them.

Next, we will outline important things that need to be tested in relation to the main functions of the website.

Searching

Searching would happen with a search keyword and then it would be filtered with a product category selection. There are four search categories: product category, price, product contents, and producer. We will assume that possible keywords could be product name and product producer. Searching with other categories would be done with checkboxes etc.

The different search category combinations need to be tested. Some test data is needed for this, so we can validate that correct products are filtered. Equivalence partitioning should be used to reduce the number of tested combinations.

Empty search should not return any products, search should not be conducted at all.

Search should work with all UTF-8 characters, but primarily it is important to work English. The application should not crash if non-latin characters, like Chinese characters are input to search. Also, special latin characters like äöüå should be accepted, as some products might have them.

Since the application uses React, it would make sense that the search is done with every change to the input field. For example, typing “a” would start showing apples, almonds etc. React is good at showing dynamic changes based on user input.

As with all user input, especially on a website, the search field input needs to be sanitized. There should be unit tests that cover possible security vulnerabilities (SQL-injection, XSS). These should also be tested manually by an expert.

These things can be tested with a combination of GUI testing and unit testing.

Shopping cart

Products are added to shopping cart by selecting them in the search view. It should be possible to add multiple instances of the same product at once in the product view.

Correct amount of the selected product should be added.

If the number of the product in the shopping cart goes to 0, it should be removed from the cart.

The amount of the product in the cart must be a positive integer.

The correct product should be added to cart. Also removing should remove the correct product.

Producer adding product (front-end)

It should be tested that correct product is added.

According to the documentation product category and contents don't have to be specified by the producer, but the other info fields should be filled. So, testing should confirm that these fields are mandatory.

We will assume that the product is added using a web form, so the input fields need to be validated. Security concerns need to be considered here as well.

The price for the product needs to be a positive decimal number. There needs to be some kind of upper limit for the price as well that can be handled by the back-end.

Functionalities not included in the test plan

The testing of the following central functions of the website are not outlined in this testing plan.

Checkout and payment are handled by a third-party software. We would at most have to do integration testing with them.

As with most online stores, it can be assumed that a website like this would have a log-in system. However, since the log-in system isn't mentioned by the documentation, we will not be writing tests related to it.

We will assume that the separate portal mentioned in the website description is not to be tested as a part of this plan, since it is a separate piece of software. The front-end app for handling products should be tested however, since we assume that is a part of the SUT.

Bug / issue categorization

We will use a basic template for describing bugs, for example:

Use case:

Expected result:

Actual result:

Revision, environment etc.

Attachments / comments

(Note: this is not a formalized template, but one we came up with.)

Bugs will be prioritized based on the severity of the bug. There will be three levels of priority:

Low

Doesn't block any actual functionality on client side or for the development team but can still be seen as faulty behaviour according to the requirements of the software or common sense. First letter not

capitalized etc. Something that would be nice-to-have but doesn't make a big difference in the bigger picture.

Medium

Causes more than a casual problem for the client or developers but doesn't block the use of the software. Medium usability issue, an error in code that doesn't block using the software but stills sends error report etc.

High

App crash, information leak, security vulnerability, or anything else that renders the website unusable or almost unusable. Something that should warrant the attention of the developers immediately. The app can't be released with a bug like this.

Code coverage

Code Coverage Percentage = (Number of lines of code executed by a testing algorithm/Total number of lines of code in a system component) * 100

Code coverage will be guaranteed by creating test cases that line up to the user scenarios. The parts of the software are tested that are most important to the actual use cases of the software. Code that changes frequently is the most important for adequate test coverage.

Adequate code coverage will be affirmed with the Coveralls tool.