Trung Truc Truong

Unit Testing in E-commerce Web Optimisation

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Media Engineering

Thesis

7 November 2016

Author(s) Title Number of Pages Date	Trung Truc Truong Unit Testing in E-commerce Web Optimisation 34 pages 7 November 2016
Degree	Bachelor of Engineering
Degree Programme	Media Engineering
Specialisation option	Mobile and .NET Programming
Instructor(s)	Iilkka Kylmäniemi, Lecturer

The purpose of this final year project was to find out the possibility of the unit testing in e-commerce web optimisation with the support of Frosmo Oy's internal tools. The tests were also designed to work under the current implementation of Frosmo's codebase.

The thesis focuses on deepening knowledge in e-commerce UI single tag development, Jasmine as a test framework and unit testing for current components. The focus was on developing understanding about unit testing by developing unit tests on two components including product tracking and conversion tracking.

Jasmine was used as a test framework as it is the main tool support by Frosmo's development team. The unit tests result in creating many different tests for many clients and spreading knowledge to the company's employees at the same time.

Keywords	Unit testing, e-commerce, Jasmine

Contents

1 Introduction

Unit testing is playing more and more important roles in software as well as web development. The idea of it is to keep the code more maintainable and documented to developers. By implementing small pieces of code with the support of many test frameworks, developer can quickly response to any changes that happened unintentionally [1].

In e-commerce web optimisation, JavaScript is one of the key programming languages that runs in the browser's engine. Since JavaScript runs in the client side, it is challenging to control client's runtime. In contrast to the server side where a specific version of backend server application can run, it is not possible to force clients to use latest version of JavaScript with the expected behaviours in the site from a developer's point of view. As JavaScript object is mutable [2], there is chances that a module is overriding another module making the code less predictable.

Jasmine, as a behaviour driven development test framework, enables the possibility to write automatic tests to developers [3].

In e-commerce web optimisation development from beginning, it is hard for developers who have little knowledge on unit testing to start the process of testing. They are usually struggling in questions such as what to test, how big a scope the tests should cover or sometimes how to write a basic test.

Spreading the knowledge of unit testing task was created at the company to discover the possibilities of unit testing. The goal of the thesis is to introduce the basic concepts of unit testing and its implementations in e-commerce web optimisation with JavaScript as the main programming language.

2 Introduction to unit testing

2.1 Overview of test driven development

When developers write code, it is challenging to know if it works and does it do what it should do. Traditionally developers have to manually check the codes by stepping through the debuggers to verify that it is working or check the output of the programmes and work out whether it is correct. This manual test usually takes a huge amount of time during the software development process. However, this traditional method also depends on how good the developer is at testing code and how confident he or she is at covering all the scopes of codes. Lately after the manual tests, it is still hard to know if it still works. More importantly, it is also hard to know if the new features that the developers just added recently did not break the codes that were made many months ago. Traditionally, programmers have designed their codes, implemented them and then tested them. Each of these phases requires a long time, sometimes even several months as seen in figure 1. [4.]

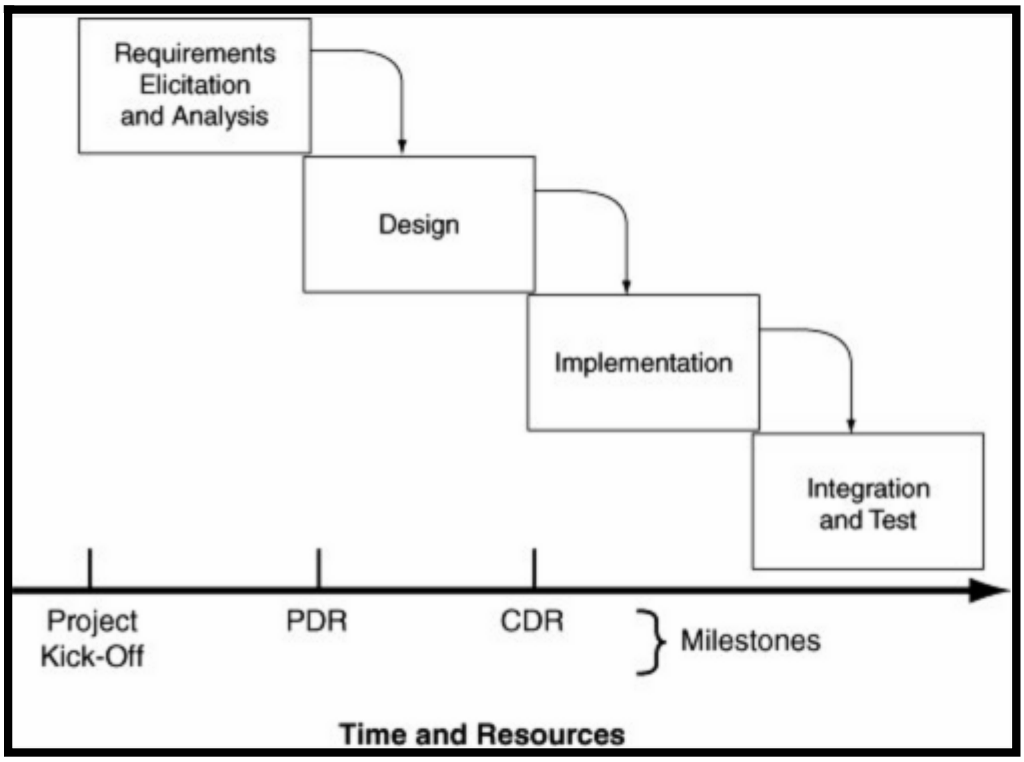


Figure 1 - Traditional Software Development Process [4]

Testing especially is every long and often ran over its scheduled time, which causes the whole project to be late. Sometimes, as there are many bugs discovered every day which need to be fixed and the project could never get finished. Testing tasks were mostly manual and they were not repeatable resulting in many buggy products. Hence, regression bugs were very common by the time. Regression bug is when something used to work and then stops working because of some new features or bug fixes. Because of the threat of regressions, manual testing takes a very long time as you have to re-test the product after every change, not just the part of the product that has been changed.

As the product grows bigger, the amount of time needed to test it grows. The other reason testing takes so long is because most of the time labelled as testing is not actually used to test; it is used to fix the bugs that were found during testing. Yet, there is often another problem when the code base grows bigger, it also grows more and more fragile which increases the chance of putting in a bug when working on some new code. [5.]

There was a high cost of change when working with the large code bases especially. The bigger the code base the more features that have been implemented and the longer it would take to add to the code a new feature that was not predicted. The figure 2 below shows that the cost of change is proportional to the size of the code base. [5.]

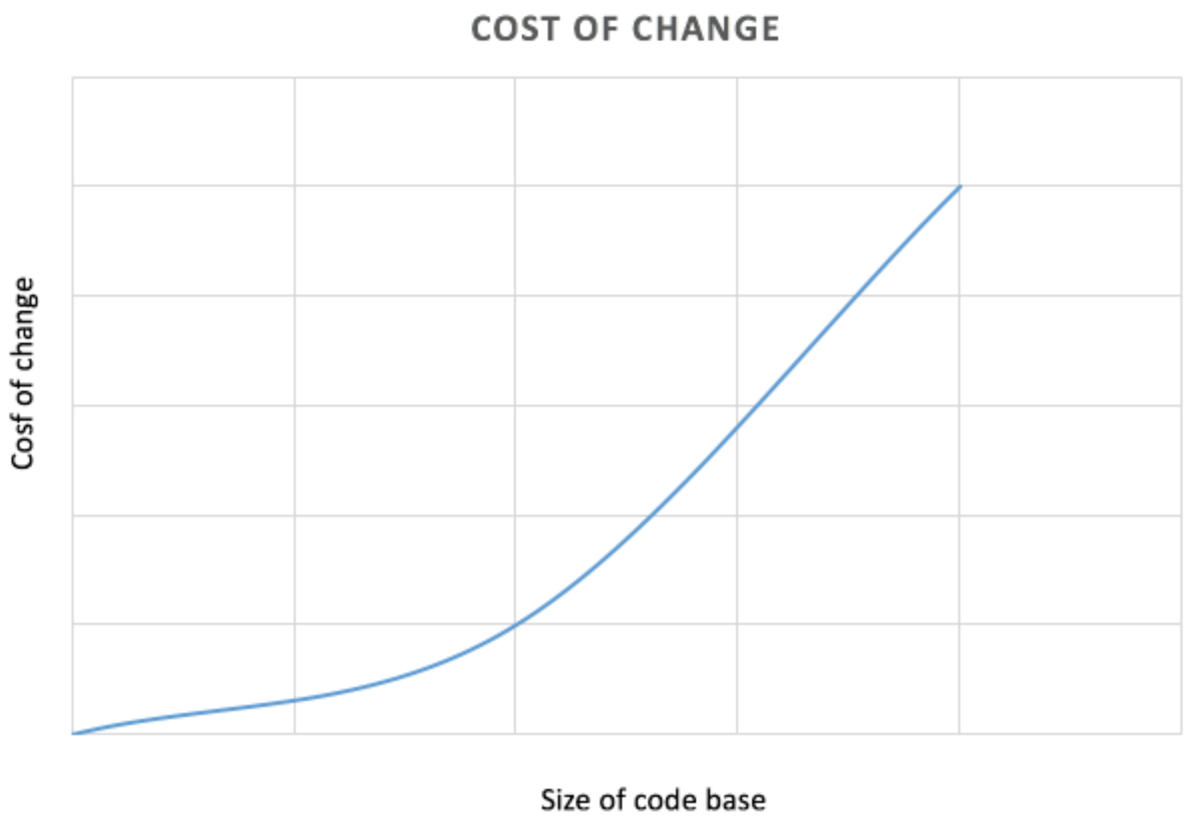


Figure 2 - Cost of change

The response of this high cost of change was trying to nail more and more things down in the design phase so that there would be less change later on. However, it rarely worked. Things could never be all decided in the design phase because there will be new design phases during the software development. Code spends most of its life in maintenance phase that is being worked on after it has been released [6]. In this case,

stopping change was the wrong thing. Instead, we need a software engineering discipline which embraces change and lowers the cost of change to software over time.
The developers including Kent Beck and the other founders of extreme programming had the desire to lower the cost of change to make software more malleable and easier to change over time. They came up with the Test Driven Design which is sometimes called test first design. [7.]

In TDD philosophy, instead of having three distinct phases including design, implement and test, the process is turned around to do test/design and implement. In this process, each of these phases are much shorter than the original phases. Instead of a phase taking several months, they take only several minutes. Developers work in smaller steps that are simple enough that they cannot get them wrong. [8.]

An application can be developed with TDD will perform the following steps as seen in figure 4 [8]:

- Write a failing test
- Make the test pass
- Refactor the code
- Repeat the above steps
- When there are no more tests that can be thought of or no more codes needed for refactoring, the process is done.

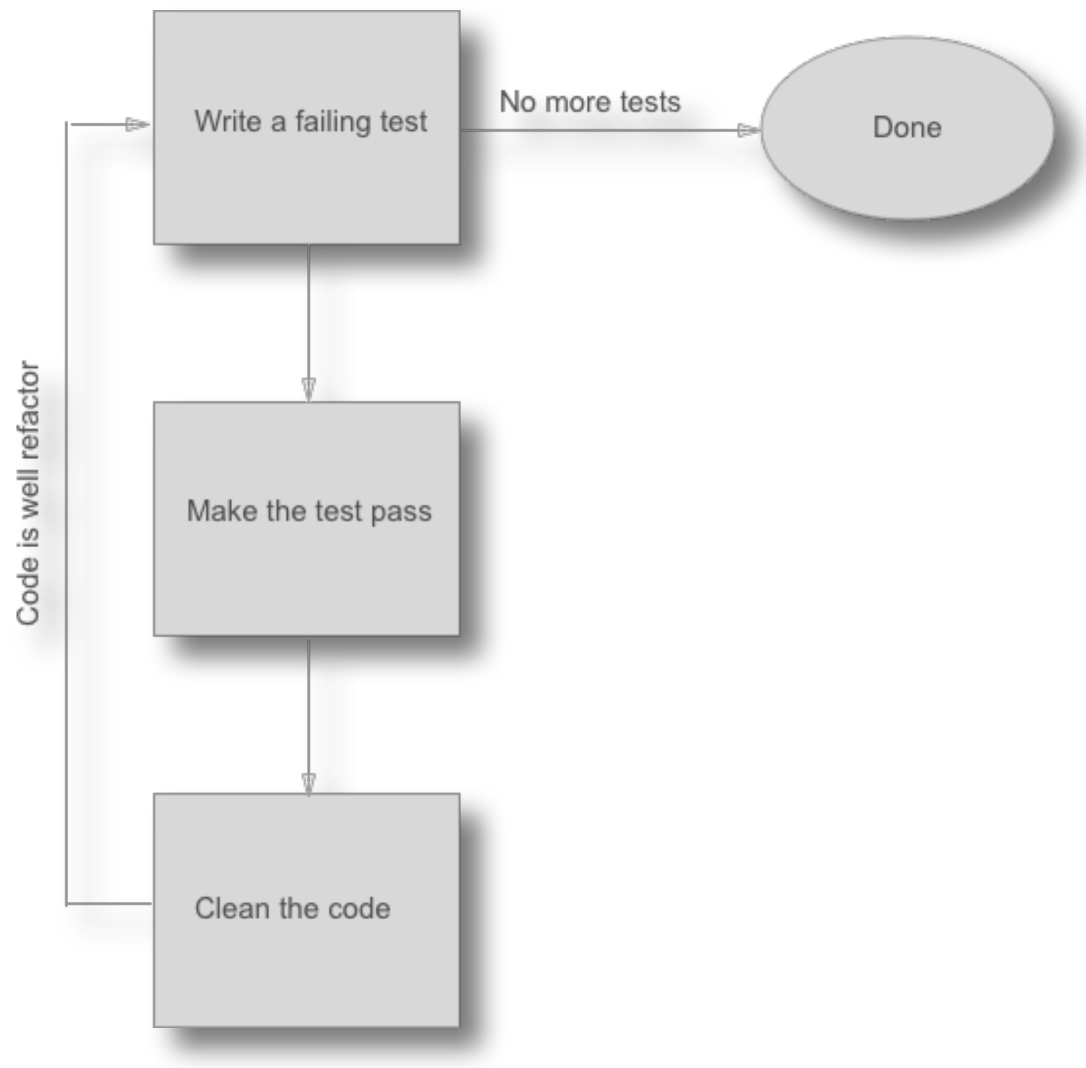


Figure 3 - TDD process

2.2 Overview of behaviour driven development

Behaviour-driven development (BDD) was introduced by Dan North to address short-comings of TDD [9]. This method focuses on the describing the behaviour of the system from the point of its stakeholders. BDD acts as a framework for collaboration and communication between developers, testers, project managers and clients.

BDD is written based on user story point of view and the criteria for the assertions is the story's behaviour [10]. The assertion criteria are described in these formats:

- Given: Initial context
- When: When event happens
- Then: Some results after that

BDD can be described as in the following figure 5

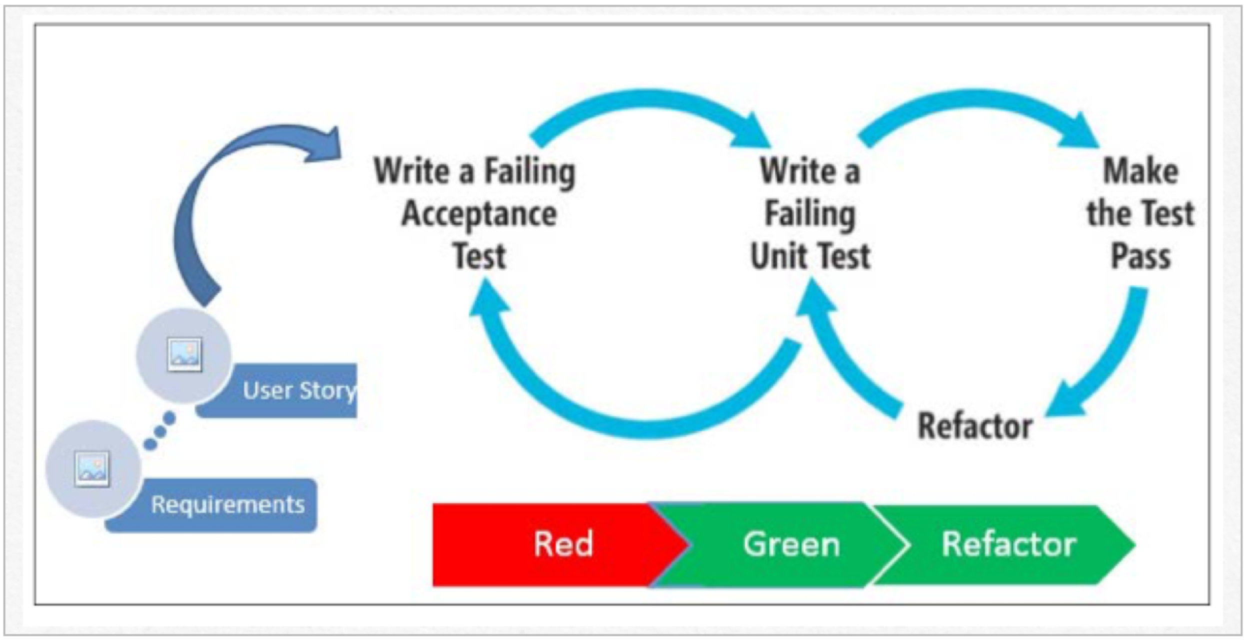


Figure 4: BDD process [11].

Test and design are one phase as we also design the system by writing tests which will use the system. This helps us to design the system which is easy to use and easy to test. A by-product of this design activity is automatic unit tests which can be easy to run to prove that the system still works. This means that developers can easily and quickly regression test the whole code base and verify that they have not broken any features that used to work. On the other hand, if developers break something, they can fix it right away within a few minutes of breaking it which is obviously the cheapest time to fix the bug because the longer the bug has been in the system the harder it is to fix. A long-standing bug is hard to fix for a couple of reasons. Firstly, new code may be relying on the buggy behaviour, so it is impossible to remove without breaking something else. Secondly, it is hard to remember why one has to write the body code and what the developer was trying to achieve when he wrote it. [11.]

Writing automatic tests and running them every time one changes the code base allows to verify whether the new code has any unexpected side effects and to fix any bugs which were introduced immediately. This method is much quicker than having a long testing phase at the end of the development. Also, it makes the developers braver as they do not need to be scared of changing the existing code in case they break something because it is easy to find out straight away and fix the bug or revert the added code. [11]

Test driven development is the practice of writing an automatic test to test the code before it is written [11].

3 Unit testing with Jasmine

3.1 Overview about Jasmine

Jasmine is a behaviour-driven development test framework which enables developers to write automated JavaScript unit tests. Behaviour-driven development is introduced by Dan North to extend test-driven development with features from domain-specific scripting language and object-oriented analysis in order to serve developers with tools and a common process to work together on software development. [3.]

3.2 Describes and specs

In Jasmine, in order to start a new test suite, the global function **describe** is called. This function takes two parameters including a string of the title of the test and a function containing all tests. In Jasmine, a test is called as a **spec**. A describe can contain one or many specs. In order to define a spec, Jasmine provides a global function **it** which takes a couple of parameters similar to that of **describe.** [3.]

Here is an example of a Jasmine test suit

```
describe("My test suit", function() {

// variables for all specs
var myConversionRate;

// Define a spec
it("Title of a Spec", function() {
// do some testing here
});

it("Another spec", function() {
// do some testing here
});

```

3.3 Expectations

In Jasmine, assertions are called as expectations. An expectation can result in either true or false and take only one argument. A matcher, which takes the expected value as its argument, is chained with an expectation. [3.]

```
expect(false).toBe(false);
```

In the above example, *expect* is an expection and *toBe* is a matcher.
In order to evaluate negative assertions, a not is then chained after *expect()*.

```
expect(true).not.toBe(false);
```

3.4 Matcher

Matchers do Boolean comparison between the actual values and the expected value by being chained after expectation and taking actual values as its arguments. [3.]
There are many matchers provided by Jasmine:

- toBe(): To compare using identity (= = =) operator

```
var a = 6
expect(a).not.toBe(6)
expect(a).toBe(5)
```

- toEqual(): Compare simple variables, literals and functions

```
expect(a).not.toEqual(6)
expect(a).toEqual(5)
```

- toMatch(): Use regular expressions for comparion

```
var str = "Frosmo Oy"
expect(str).toMatch(/Frosmo/)
expect(str).not.toMatch(/Orange/)
```

- toBeDefined(): Check if a variable or a function is defined.

```
expect(a).toBeDefined(); // Will pass
expect(a).not.toBeDefined(); // Will not pass
```

- toBeNull(): Check if variable is null

```
expect(a).not.toBeNull();
expect(null).toBeNull();
```

- toBeTruthy(): Check if an expression or variable is true

```
var b = true;
var c = false;
expect(c).not.toBeTruthy();
expect(add(1,2)).toBeTruthy();
expect(b).toBeTruthy();
```

- toBeFalsy(): Check if an expression or variable is false

```
expect(add(0,0)).toBeFalsy();
expect(b).not.toBeFalsy();
```

- toContain(): Check if a collection such as an array has an item or not

```
var frosmonaults = ["truc", "omid",  "catarina"]
expect(frosmonaults).toContain("truc")
```

- toBeLessThan(): To compare numbers in mathematics

```
expect(3).toBeLessThan(4)
```

- toBeGreaterThan(): This is the negative version of toBeLessThan()
- toBeCloseTo(): Check if a number if close to another number. This takes 2 arguments, the first is the number and the second is decimal precision.

```
expect(5.1234).toBeCloseTo(5.1, 1); // Pass
expect(5.1234).not.toBeCloseTo(5.1, 2); // Pass
```

- toThrowError(): Check if a specific exception was thrown from a function

```
var frsm = function() {
  throw new TypeError("A custom Exception from frsm");
};
expect(frsm).toThrowError("custom");
```

- jasmine.any(): If we are not aware of the actual value but know the type, this can help

```
expect([12,456]).toEqual(jasmine.any(Array));
expect({}).toEqual(jasmine.any(Object));
expect(90245).toEqual(jasmine.any(Number));
```

- jasmine.objectContaining(): Compare partially in key-value cases

```
var employee = {
    name: "Truc",
    age: 99,
    department: "Customer Service Team"
}
expect(employee).toEqual(jasmine.objectContaining({
    name: "Truc"
}));
expect(employee).toEqual(jasmine.objectContaining({
    age: jasmine.any(Number)
}));
```

3.5 Spies

A Spy is an emulation of an object or a function regardless of the function/object being defined or not. In Jasmine, a spy can stub any function and detect any function call to that function as well as the arguments being passed on. A spy can stub function but can exist only within it and describe. A spy function needs to be public in order to be spied on. [3.]

There are some special matchers for spies as in the following:

- toHaveBeenCalled(): Return true if the spy is called
- toHaveBeenCalledWith(): Return true if the spy is called with arguments

```
describe("A spy", function() {
  var foo, bar = null;

  beforeEach(function() {
    foo = {
      setBar: function(value) {
        bar = value;
      },
      getBar: function() {
        return bar;
      }
    };
    spyOn(foo, 'setBar');
    foo.setBar(123);
    fetchedBar = foo.getBar();
  });

  it("tracks that the spy was called", function() {
    expect(foo.setBar).toHaveBeenCalled();
    expect(foo.setBar).toHaveBeenCalledWith(123);
  });

  it("should not affect other functions", function() {
    expect(bar).toEqual(123);
  });

  it("when called returns the requested value", function() {
    expect(fetchedBar).toEqual(123);
  });

})
```

- toCallThrough(): Since Jasmine replaces the actual implementation of the function on which spyOn() is called, therefore if there is a need to get an actual output of a function we have to chain spyOn() with and.callThrough().

```
spyOn(frosmo, "test").and.callThrough();
```

- returnValue(): Return a specific value from a function after creating a spy by chaining and.returnValue().

```
spyOn(site.utils, 'getHref').and.returnValue("http://domain.com");
```

- callFake(): Provide own implementation of function because actual implementation might take time to execute or can be some other reasons. In this case we can chain and.callFake(). This will take a function as an argument.

```
spyOn(site.utils, "ajaxCallToBasket").and.callFake(function(grade){
  return true;
});
```

- throwError(): When there is a need for a function to throw an error, we can chain a function with and.throwError().

```
spyOn(site.utils, "getValue").and.throwError("Service is down");
expect(site.utils.getValue).toThrowError("Service is down");
```

3.6 HTML fixtures

Test fixtures provide basic states on which the test is running such as a piece of HTML, the definition of an object and so on. For example, if there is a need to tests on the form submission, the HTML form has to be available when the test is running. [3.]

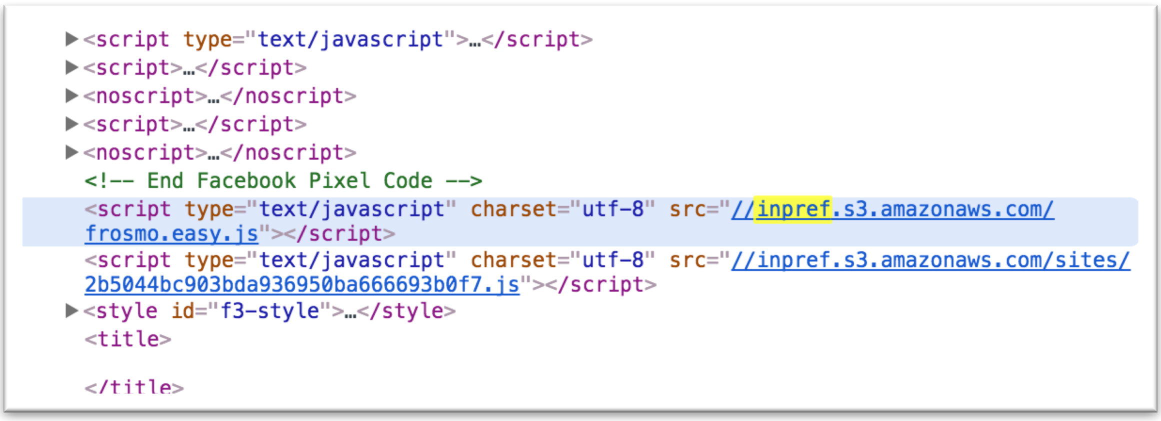
The HTML codes that contain the HTML form is a fixture:

```
beforeEach(function() {
    $('body').append('<form id="example-form"></form>');
});
```

4 Introduction to single tag UI optimisation in e-commerce

4.1 Overview of single tag UI optimisation development

The single tag platform is a web UI development solution for improving website functionality and personalizing online user experience. The platform works in the front end through JavaScript via tags that are placed directly in the web page HTML code as seen in figure 5. [12.]



```
▶<script type="text/javascript">…</script>
▶<script>…</script>
▶<noscript>…</noscript>
▶<script>…</script>
▶<noscript>…</noscript>
 <!-- End Facebook Pixel Code -->
 <script type="text/javascript" charset="utf-8" src="//inpref.s3.amazonaws.com/
 frosmo.easy.js"></script>
 <script type="text/javascript" charset="utf-8" src="//inpref.s3.amazonaws.com/sites/
 2b5044bc903bda936950ba666693b0f7.js"></script>
▶<style id="f3-style">…</style>
 <title>

 </title>
```

Figure 5: Third party script being inserted to customer's website.

Front-end-based implementation brings two major advantages:

- The majority of the development happens in the visitor's browser, rather than in the back end, so making changes and implementing new functionality is very fast. This means less server technology and fewer resources are needed, which makes UI development inexpensive. [12.]
- JavaScript can modify anything on a web page [13], so we can create flexible customizations that allow addressing specific pain points, such as shopping cart abandonment, customization of product recommendations, or need to change the UI quickly and temporarily for a campaign.

4.2 Basic concepts

Web UI development generally means designing a website's user interface so that a website visitor can find information and complete relevant tasks easily and intuitively These are some of the most important concepts related to UI development that are popularly used in this field:

- Personalization: Making real-time changes to a website based on the attributes and behaviour of the visitor. Personalization adapts the options presented to the visitor to guide them through a funnel. Technically, web personalization is implemented by applying specific changes to the website based on visitor segmentation. The goal of personalization is to provide a more relevant user experience and, as a result, drive conversion and create revenue. Personalization can also be used to cross-sell or upsell products and create brand loyalty through engagement. To do this effectively, it is needed to understand visitors' interests and motivations. [14.]

- Conversion: Getting website visitors to do what the shop owner would like them to do. This can be buying products or, for example, signing up for newsletters, downloading a brochure or white paper, or watching a video. What is defined as a conversion depends on separate business goals. [15.]
- Conversion rate: The percentage of all visitors that actually make a conversion. The conversion rate is constantly increasing and decreasing due to seasonal changes, marketing campaigns, and sales. If only counting the number of conversions, it may stay the same even though the traffic on the website has increased, meaning that an increasing number of visitors see the content but a relatively decreasing number responds to it. [15.]
- Conversion tracking: The method used to count conversions and assign them to displayed or clicked content [16].
- Segmentation: Grouping website visitors based on their behaviour, location, or other variables to enable effective adaptive content. It is possible to define a set of rules that place a visitor into one or more segments – or remove a visitor from them. Segments are used to target specific types of visitors with content that meets their interests. Content variations displayed to a visitor depend on the segment that the visitor is in. [17, 25-34.]
- Adaptive (or dynamic, or smart) content: Content on a website or in an email that changes based on the visitor's behaviour, location, and other variables, creating a customized user experience. One of the most common applications of adaptive content is to provide product recommendations in a web store based on the products the visitor has recently viewed. [18.]
- A/B testing: The most common way to test optimized or personalized content. In A/B testing, visitors are split into two groups (A and B, respectively) and show different content variations to those groups. By using analytics tool, project planners are able to see which content variation performed the best, that is, resulted in the highest conversion rate or average order value. [19.]

5 Unit testing in action

5.1 Testing environment

5.1.1 Operating system

Jasmine is cross platform for testing JavaScript code where there is browser. Therefore, it can be used on top of any operating system ranging from Linux, Windows to MacOS. In the scope of this project, Linux CentOS Linux version 7.2 is the system of choice. In this scope of this project, there are usages of Frosmo's own tool and code modules. These includes some automation tools and utility helper functions.

5.1.2 Text editor

Sublime Text 3 will be the main code editor for this project to implement unit tests. Sublime Text supports Package Control which allow developers to install additional features to support the workflow such as code completion for Jasmine or JavaScript. [20.]

5.1.3 Jasmine test framework

The previous sections have introduced about Jasmine test framework and a number of its functionalities. Jasmine does not depend on any JavaScript framework. The Jasmine version of choice for this project is 2.0. With the current setup, the folder structure would look like in figure 6.

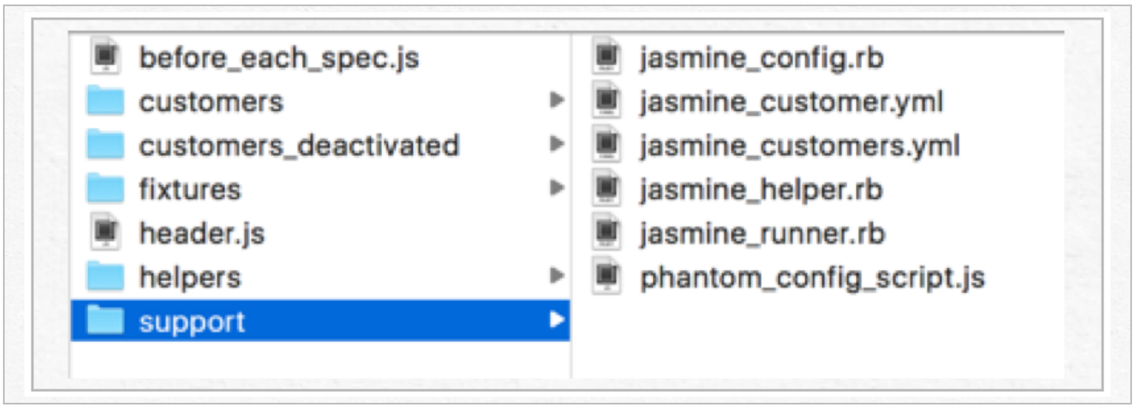


Figure 6: Jasmine folder structure.

Each folder in the figure above plays different roles in implementing tests:

- Support: Containing Jasmine's general configuration when running tests
- Helpers: Containing special supporting files or configurations for projects that require.
- Fixtures: Containing HTML fixtures that Jasmine will use when there is a need for testing DOM elements.
- Customers: Containing set of tests for customer's projects.

5.2 Unit testing for product tracking

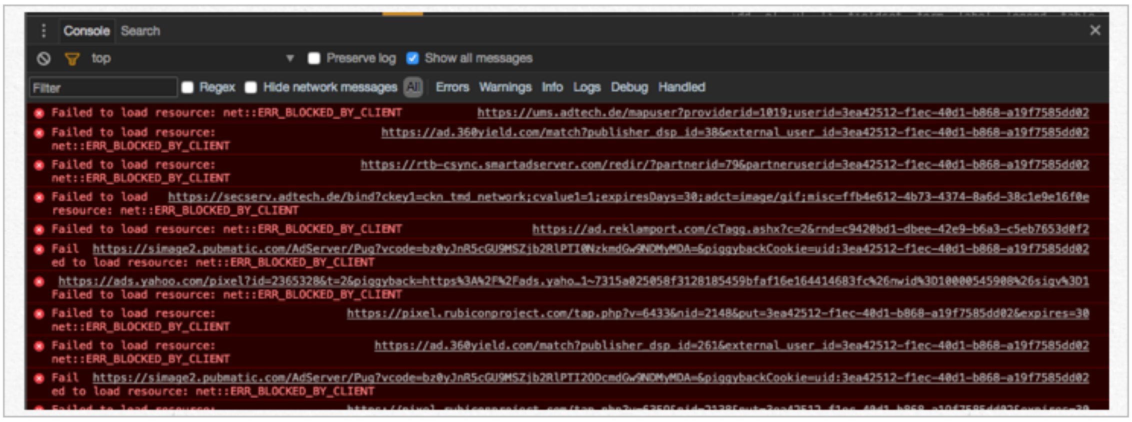
5.2.1 Product tracking concept

Gathering data is considered as one step in UI development in order to get to know more about customer's behaviours on the site. There are some data analytic tools, such as Google Analytics, that can help to gather user's data such as site traffic, traffic source, page views, new and returning visitors, session duration, and bounce rate.

The more complete the data is, the better to understand how visitors behave and what engages, motivates and attracts them. Product tracking contributes significantly into this data pile. Product tracking provides basic information on the products the website has, such as product names, categories, prices and a number of product page visits each product gets. One common use of product tracking data is product recommendation that offers a set of products based on visitor's interests. [21.]

5.2.2 The unit test for product tracking

The aim of this unit test is to verify the product tracking is working correctly in its own scope and sending correct data to the backend such as correct product names, prices, categories. Since the data source is dynamic and taken directly from the customer's product page, developers cannot always assume the input data types to be valid without validating them first before saving to the database. Besides, if developers are not able to handle their implementations properly, they might send many errors in customer's site and break some of the functionalities of the original website as seen in figure 7.



```
Failed to load resource: net::ERR_BLOCKED_BY_CLIENT    https://ums.adtech.de/mapuser?providerid=1019;userid=3ea42512-f1ec-40d1-b868-a19f7585dd02
Failed to load resource:                               https://ad.360yield.com/match?publisher_dsp_id=38&external_user_id=3ea42512-f1ec-40d1-b868-a19f7585dd02
net::ERR_BLOCKED_BY_CLIENT
Failed to load resource:                               https://rtb-csync.smartadserver.com/redir/?partnerid=79&partneruserid=3ea42512-f1ec-40d1-b868-a19f7585dd02
net::ERR_BLOCKED_BY_CLIENT
Failed to load   https://secserv.adtech.de/bind?ckey1=ckn_tmd_network;cvalue1=1;expiresDays=30;adct=image/gif;misc=ffb4e612-4b73-4374-8a6d-38c1e9e16f0e
resource: net::ERR_BLOCKED_BY_CLIENT
Failed to load resource: net::ERR_BLOCKED_BY_CLIENT    https://ad.reklamport.com/cTagg.ashx?c=2&rnd=c9420bd1-dbee-42e9-b6a3-c5eb7653d0f2
Fail  https://simage2.pubmatic.com/AdServer/Pug?vcode=bz0yJnR5cGU9MSZjb2RlPTI0NzkmdGw9NDMyMDA=&piggybackCookie=uid:3ea42512-f1ec-40d1-b868-a19f7585dd02
ed to load resource: net::ERR_BLOCKED_BY_CLIENT
 https://ads.yahoo.com/pixel?id=2365328&t=2&piggyback=https%3A%2F%2Fads.yaho…1~7315a025058f3128185459bfaf16e164414683fc%26nwid%3D10000545908%26sigv%3D1
Failed to load resource: net::ERR_BLOCKED_BY_CLIENT
Failed to load resource:                               https://pixel.rubiconproject.com/tap.php?v=6433&nid=2148&put=3ea42512-f1ec-40d1-b868-a19f7585dd02&expires=30
net::ERR_BLOCKED_BY_CLIENT
Failed to load resource:                               https://ad.360yield.com/match?publisher_dsp_id=261&external_user_id=3ea42512-f1ec-40d1-b868-a19f7585dd02
net::ERR_BLOCKED_BY_CLIENT
Fail  https://simage2.pubmatic.com/AdServer/Pug?vcode=bz0yJnR5cGU9MSZjb2RlPTI2ODcmdGw9NDMyMDA=&piggybackCookie=uid:3ea42512-f1ec-40d1-b868-a19f7585dd02
ed to load resource: net::ERR_BLOCKED_BY_CLIENT
```

Figure 7. Errors that are visible in the console window

Therefore, it is necessary to have a set of unit tests in order to guide developers to the right way and to make the project more maintainable. Also, having unit tests for product tracking can help avoid unpredictable changes coming from the customer's site without any developer's knowledge. For example, in case the customer changes the structure of the product page it is easier to find out where the error happens in order to debug it. Also, if there are many developers taking part in different parts of the project, the tests will help to make the codebase more consistent by forcing them to write functions that can pass the tests.

5.2.3 How product tracking module works

Product data such as product name, code, category, image are taken from the sites via different methods.

Product data includes:

- Product name: Define the name of the product
- Product ID: Define product's ID to identify product
- Product category: Define which categories the product belongs to
- Product image: Define product's preview image
- Product price: Define the current price of the product
- Product discount price: Define the current discount price of the product
- Product promotion label: Define other properties of the products such as product availabilities, product's colour.

These product's properties will be put in an JSON object. After that the main method 'handleItem' will verify and send data to backend.

```javascript
/**
* Product Tracking
* Save product data to back end.
*
*/
function _trackProduct() {
    easy.addExceptionHandling(function (){
        var product = {};
        product.myProductName = _getProductName();
        product.myProductId = _getProductCode();
        product.myProductCategory = _getProductCategory();
        product.myProductImage = _getProductImage();
        product.myProductPrice = _getProductPrice(_getProductNew-
Price(), _getProductOldPrice());
        product.myProductUrl = _getProductUrl();
        product.myProductDiscountPrice = _getProductDiscountPrice(_getProduct-
NewPrice(), _getProductOldPrice());
        product.myProductPromotionLabel = my.JSON.stringify({
            product_stock_availability: _getProductStockAvailability(),
            boot_guarantee: _isBootGuarantee()
        });
        easy.dataLayer.handleItem(product);

    }, {
        code: easy.ERROR_SITE_PRODUCT
    })();
}
```

5.2.4 Scenarios

Assuming that we already have product tracking functionality up and running and there is no test written for it yet. The product tracking collects data from the page and from global scope variables from the page.

The unit test is needed in this case to make the test file remain its consistency no matter how many developers are working on it the same time.

The test can be initialized by creating a new test file in the specs folder namely 'productTrackingSpec.js'. The reason why the file name ends with Spec is because when running tests, Jasmine will include all JavaScript files that contain the Spec keyword in their file names in the Spec folder.

The test file will begin with the following structure.

```
describe("product tracking", function () {

    beforeEach(function () {
        // Define condition before running tests
    });

    it("should send product data", function () {
        // Some methods to handle productData are called here.
        expect(easy.dataLayer.handleItem).toHaveBeenCalledWith(productData);
    });

    afterEach(function (){
        // Clean up after the tests.
    });

});
```

5.2.5 Applying unit tests

In order for the test to work properly it is needed to proper environment that is close to the real environment. These can be provided within the 'beforeEach' function. In Jasmine, 'beforeEach' is called before executing any assertion.

First, there is a need to declare required values needed for the tests to run.

```
describe("customer1 productTracking", function () {

    var easy, site, utils, page, productConfigs, expectedResults, extra-
Data, productParser;
```

```
    easy = frosmo.easy;
    site = frosmo.site = frosmo.sites.customer1;
    utils = frosmo.sites._utils;
    page = window.jQuery('<div></div>').html(window.fixtures.customer1_prod-
uct_page)[0];
    productConfigs = site.productTracking.configs;

    expectedResults = {
        id: 'VRSCVDA010014',
        name: 'Bayan Kol Saati',
        urn: '389',
        type: 'bayan saat',
        category: 'saat',
        subCategory: 'bayan',
        price: '3689.00',
        discountPrice: '1844.50' ,
        manufacturer: 'Versace',
        gender: 'Bayan',
        sale: true,
        color: 'Siyah',
        case: 'Yuvarlak',
        strap: 'Deri',
        glass: 'Safir' ,
        style:  'Klasik Saatler',
        tech: 'Analog'
    };
    productParser = utils.productParser(page, productConfigs);

})
```

In the above codes, we can see that some external modules are called to prepare for the tests. Also, DOM fixture is also created by jQuery's 'html' method which sets the HTML content of the element based on the html string that it receives as the input. Besides there are some results that we expect the tests to validate. [22.]

- 'site' is the predefined namespace for the current testing customer
- 'productConfigs' is a module defined in product tracking implementation to define how to retrieve data from the page and how to validate them before returning.
- 'utils' is a module written by the enterprise to levitate part of the implementation for developers.
- 'productParser' is a module to process data based on the values it receives which are the DOM page and configurations

In Jasmine tests, it is not possible to get same value via global methods such as 'window.location.href' in the testing environment and in production environment. Therefore, it is needed to mock the fake value via a helper function from Jasmine 'spyOn'. Since Jasmine tests run on localhost, the method of getting current URL will return a localhost address instead of the real URL. In this case we can use ‘spyOn’ to the function that gets the current URL by the following code

```
beforeEach(function () {
    spyOn(site.utils, 'getCurrentUrl').and.returnValue(customUrl);
});
```

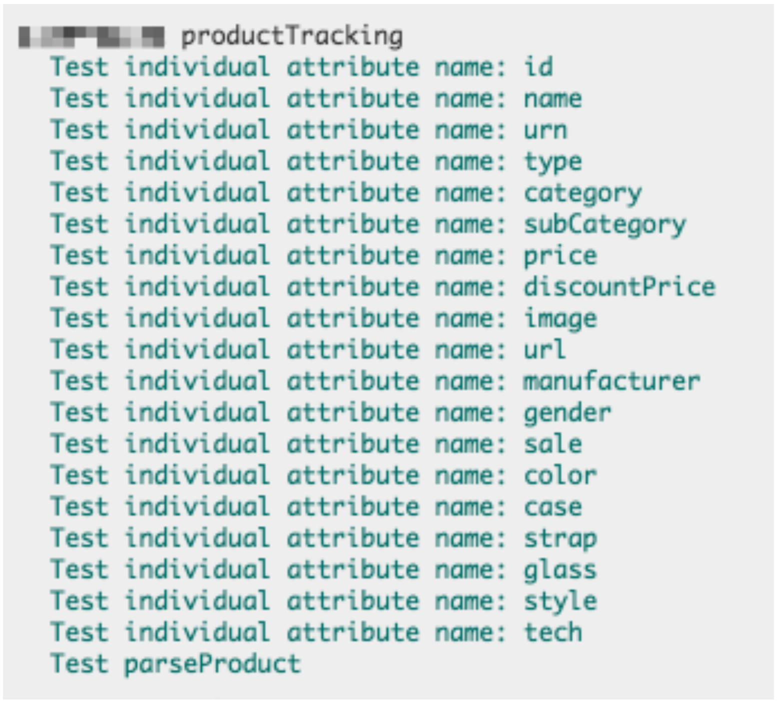
‘spyOn’ helper function will replace all the function call to ‘site.utils.getCurrentUrl’ with its own implementation and force it to return our custom url.

The actual tests are written within Jasmine’s ‘it’ functions

```
easy.utils.each(productConfigs, function (attrData, attrName) {
it("Test individual attribute name: " + attrName, function () {
    var foundData;

    foundData = productParser.parseProductAttribute(attrData, attrName);
            expect(foundData).toEqual(expectedResults[attrName]);
        });
    });
});
})
```

The code above actually initializes many tests at once by using the ‘each’ iterator. The loop will go through each method in ‘productConfigs’ and test what it returns with the accordingly expected values.

In order to see the result in the browser, the tests are run and the browser returns the result as seen in the following figure:



```
[redacted] productTracking
  Test individual attribute name: id
  Test individual attribute name: name
  Test individual attribute name: urn
  Test individual attribute name: type
  Test individual attribute name: category
  Test individual attribute name: subCategory
  Test individual attribute name: price
  Test individual attribute name: discountPrice
  Test individual attribute name: image
  Test individual attribute name: url
  Test individual attribute name: manufacturer
  Test individual attribute name: gender
  Test individual attribute name: sale
  Test individual attribute name: color
  Test individual attribute name: case
  Test individual attribute name: strap
  Test individual attribute name: glass
  Test individual attribute name: style
  Test individual attribute name: tech
  Test parseProduct
```

Figure 8. Expected test result from product tracking

As can be seen from the figure above, the code looped through every single configuration of the 'configs' module and test it with the expected values that we defined. This test returns correctly what we expect with green lines.

5.3 Unit test for conversion tracking

5.3.1 How conversion tracking concept

As mentioned in product tracking, gathering data is very important in making better user interface for customers. Conversion tracking involves in tracking what items that customers purchased and their quantities. This helps the company to figure out which factors affect the most on the decision to buy a product from customers and can make adjustment through the user interface. While product tracking gives the site owner a picture of which products customers view the most, conversion tracking delivers what customers actually end up buying instead. Conversion tracking is one of the key factors in helping to define if the business is going well. [15.]

The conversion tracking is done by tracking the customer's baskets in basket page. In that page, products in the customer's basket will be saved in either local storage or to the back-end. After the customers have made a successful payment, data about the purchased products that were saved before will be sent to the central server

5.3.2 Unit testing for conversion tracking

Conversion tracking in this topic is implemented by ordinary method of adding items in customer's basket page to local storage along with total order value. These items are sent to the backend when users have completed their transactions. The conversion tracking is implemented as below

```javascript
/*global easy, site*/
site.conversionTracking = (function () {
    var dom = document;
    var sessionStorage = easy.store;
    var initialized = false;

    /**
     * Send product conversion to back-end
     * @param  {DOM} $html [HTML DOM]
     */
    function _sendProductConversion($html) {
        var products = sessionStorage.get('productBasket');
        var  orderTransactionTotal =  sessionStorage.get('transactionTotal') ? sessionStorage.get('transactionTotal') : null;
        // Exit if there is no products
        if (!products || !products.length){
            return;
        }
        // Handling (sending) the transaction
        easy.addExceptionHandling(function () {
            easy.dataLayer.handleItem({
                transactionId: _getTransactionID($html),
                transactionProducts : products,
                transactionTotal:  orderTransactionTotal
            });
            _clearBasketProducts();
            _clearBasketAddToCartProducts();
        }, {
            code: easy.ERROR_CONVERSION_SEND
        })();
    }

    /**
     * Get transaction ID from DOM
     * @param  {DOM Node} $html [the current product node]
     * @return {String}
     */
    function _getTransactionID($html) {
        return $html.querySelector('.main a') ? $html.querySelector('.main a').innerHTML : $html.querySelector('.main p').innerHTML.split(" ")[2].replace(/[^0-9]+/g, "");
    }


    /**
     * Get product data from the page and save to localStorage with easy store
     * @param  {HTML DOM} $html [the DOM]
     */
    function _saveProductsToLocalStorage($html) {
        var products = [];
        var addToCartProducts = [];
        var productList = $html.querySelectorAll('table#shopping-cart-table tbody tr');
        var productPageList = sessionStorage.get('addToCartProducts') || [];
        // Clear all products if basket page is empty
        if (!productList.length) {
            _clearBasketProducts();
            _clearBasketAddToCartProducts();
            return;
        }
        easy.addExceptionHandling(function () {
            easy.utils.each(productList, function (product) {
                var comparedId = _getProductId(product);
                products.push({
                    id       : comparedId,
                    name     : _getProductName(product),
                    category : _getProductCategory(product),
                    price    : _getProductPrice(comparedId, product),
                    quantity : _getProductQuantity(product)
                });
                // Reduce the size addToCartProducts in sessionStorage
                var existProduct = easy.utils.find(productPageList, function (product) {
                    return product.id === comparedId;
                });
                if (existProduct) {
                    addToCartProducts.push(existProduct);
                }
            });
            var  transactionTotal = _getTransactionTotal($html);
            sessionStorage.set('productBasket', products);
            sessionStorage.set('transactionTotal', transactionTotal);
            sessionStorage.set('addToCartProducts', addToCartProducts);
        }, {
            code: easy.ERROR_PRODUCTS_GET_ERROR_CONVERSION
        })();
    }

    /**
     * Get transaction total from DOM
     * @param  {DOM Node} $html [the current product node]
     * @return {String}
     */
    function _getTransactionTotal($html) {
        return site.utils.formatCurrency($html.querySelector('.mnm-grand-total span.price').innerHTML);
    }

    /**
     * Get Product id from DOM
     * @param  {DOM Node} $html [the current product node]
     * @return {String}
     */
    function _getProductId($html) {
        return $html.querySelector('a.product-image').getAttribute("title").split(" ")[0];
          }

          /**
           * Get current product name from DOM
           * @param  {DOM Node} $html [the current product node]
           * @return {String}
           */
          function _getProductName($html) {
              return $html.querySelector('a.product-image').getAttribute("title");
          }

          /**
           * Get current product category from DOM
           * @param  {DOM Node} $html [the current product node]
           * @return {String}
           */
          function _getProductCategory($html) {
              return $html.querySelector('a.product-image').getAttribute("title").split(" ")[1];
          }

          /**
           * Get current product quantity from DOM
           * @param  {DOM Node} $html [the current product node]
           * @return {String}
           */
          function _getProductQuantity($html) {
              return parseInt($html.querySelector('input.qty').getAttribute("value"), 10);
          }

          /**
           * Get current price since the price showing in the basket page is incorrect
           * current price is taken from sessionStorage saved in previous pages
           * @param  {String} id    [id of the current product]
           * @param  {DOM Node} $html [HTML node of the current product]
           * @return {String}       [Product price]
           */
          function _getProductPrice(id, $html) {
              var basketPrice = site.utils.formatCurrency($html.querySelector('span.cart-price span.price').innerHTML);
              var productList = sessionStorage.get('addToCartProducts');
              // Return normal basket price if we didn't get any price from previous product pages;
              if (!productList) {
                  return basketPrice;
              }
              // Check the real price we stored from previous page;
              var existProduct = easy.utils.find(productList, function (product) {
                  return product.id === id;
              });
              return existProduct ? existProduct.price : basketPrice;
          }

          /**
           * Clear products that users to cart in product page
           */
          function _clearBasketAddToCartProducts() {
              sessionStorage.remove('addToCartProducts');
          }

          /**
           * Clear products in baskets page that is supposed to be empty
           */
          function _clearBasketProducts() {
              sessionStorage.remove('productBasket');
          }

          /**
           * Save product when users click on add to cart in Product Page
           * @param  {DOM} $html [HTML DOM of the current product page]
           * @param  {function} varOriginalClickEvent [Original click event]
           * @param  {click event} mouseEvent [Click event by default]
           */
          function _saveProductWhenAddToCartIsClicked( $html, varOriginalClickEvent, mouseEvent) {
              var productList = sessionStorage.get('addToCartProducts') || [];
              var existProduct = null;
              var comparedId = $html.querySelector(".product-definitions h2").innerHTML.split(" ")[0];
              // Check if we already have the product in the storage already
              existProduct = easy.utils.find(productList, function (product) {
                  return product.id === comparedId;
              });
              // Save product if not exists in our storage
              if (!existProduct) {
                  var rulePriceSelector = $html.querySelector(".price-info .special-price.has-rule-price .rule-price");
                  var priceSelector = rulePriceSelector ? rulePriceSelector : $html.querySelector(".price-info .special-price .price");
                  var currentPrice =  site.utils.formatCurrency(priceSelector.innerHTML);
                  productList.push({
                          id : comparedId,
                          price: currentPrice
                      });
                  sessionStorage.set('addToCartProducts', productList);
              }
              // Return normal click event;
              varOriginalClickEvent();
          }

          /**
           * Save product when users click on add to cart in Category Page
           * @param  {function} varOriginalClickEvent [Original click event]
           */
          function _saveProductWhenAddInCatPage(varOriginalClickEvent) {
              var target = event.srcElement;
              var parent = target.parentElement.parentElement;
              parent = parent.className!=="product-info" ? parent.parentElement.parentElement : parent;
              var myPrice = site.utils.formatCurrency(parent.querySelector('p.special-price span.price').innerHTML);
              var myId = parent.querySelector('h2.product-name a').innerHTML.trim().split(" ")[0];
              var productList = sessionStorage.get('addToCartProducts') || [];
              productList.push({
                  id : myId,
                  price: myPrice
              });
              sessionStorage.set('addToCartProducts', productList);
              // Return normal click event;
              varOriginalClickEvent();
          }

          function init() {
              easy.events.on(easy.EVENT_DOM_READY, easy.addExceptionHandling(function (){
                  // Handle add to product cart in product page
                  if (site.pages.isProductPage()) {
                      var btn = dom.querySelector('button.btn-cart');
                      var varOriginalClickEvent = btn.onclick;
                      btn.onclick = null;
                      easy.domEvents.on(btn, 'click', _saveProductWhenAddToCartIsClicked.bind(this, dom, varOriginalClickEvent));
                  }
                  // Handle add to product cart in category page
                  if (site.pages.isCategoryPage()) {
                      var addToCartBtns = dom.getElementsByClassName("btn-cart");
                      easy.utils.each(addToCartBtns, function (btn) {
                          var varOriginalClickEvent = btn.onclick;
                          btn.onclick = null;
                          easy.domEvents.on(btn, 'click', _saveProductWhenAddInCatPage.bind(this, varOriginalClickEvent));
                      });
                  }
                  // Handle in basket page
                  if (site.pages.isBasketPage()) {
                      _saveProductsToLocalStorage.apply(this, [dom]);
                  }
                  // Send conversion in checkout successful page
                  if (site.pages.isCheckoutPage()) {
                      _sendProductConversion.apply(this, [dom]);
                  }
              }));
              initialized = true;
          }

          return {
              init: init,
              initialized: initialized,
              dom: dom
          };
      })();
```

In actual entrance for the conversion tracking is in the ‘init’ method. In the ‘init’ method, it can be seen that the codes have been implemented with DOM event listener. When DOM is ready the actual codes will be executed. Since all the information about the product cannot be taken in basket page, it is necessary to record what users have added to their basket in previous pages including product page and category page. In basket page, the product is then saved to local storage for later use. If user has made the purchase successfully, the private method ‘_sendProductConversion’ will handle the data sending to our backend.

It is also important to notice that in this file, there are many ‘apply’ methods implemented. These ‘apply’ guarantee that when Jasmine is running, the conversion tracking will get proper scopes. [23.]

From the conversion tracking implementation, there are many unit tests to cover. Among those tests, two most important tests are

- Product data should be saved correctly to local storage in basket page
- Product data should be sent to the back end when transaction is completed successfully.

5.3.3 Applying unit tests

Conversion tracking unit tests should follow the following basic structure.

```
describe("customer1 conversionTracking", function () {

    // Declaration of necessary properties

    beforeEach(function () {

    });

   it("Should save correct data to local storage in basket page", func-
tion (){
        // Tests assertion will be here
    });

    it("Should send data from local storage to backend", function (){
        // Tests assertion will be here
    });

    afterEach(function () {

    });

});
```

In the declaration of properties, the test needs a proper environment to run and, therefore, external modules should be included

```
describe("customer1 conversionTracking", function () {

    var easy, site, utils, page, oldSiteValue,expectedResult;

    easy = frosmo.easy;
    site = frosmo.site = frosmo.sites.saatvesaat;
    utils = frosmo.sites._utils;
    page = window.jQuery('<div></div>').html(window.fixtures.basketPage)[0];
    successPage = $('<div></div>').html(window.fixtures.success_pay-
ment_page)[0];
    //...

});
```

it can be seen that with fixtures for basket page and successful payment page are created. These will be used later in separated tests.

The first test will verify if the products in the basket pages will be saved correctly to local storage. In order for the products to be saved there are several conditions that need to be thought of before making the test

- Current page is basket page, not in other pages.
- The method of handling saving products to local storage needs to be spied on with ‘spyOn’ function.

- The conversion tracking is only triggered after DOM ready event takes place.

Since the Jasmine tests run locally, there is not sufficient information for Jasmine to identify which current page it is testing on. In order to reduce the work for developers, Jasmine provides ‘SpyOn’ helper function. In this case, ‘spyOn’ function will be used to page identifier methods which will basically replace current implementation of the page and force it to return defined value.

```
// Always return false when checking if current page is Product Page
spyOn(site.pages, 'isProductPage').and.returnValue(false);
// Always return false when checking if current page is Category Page
spyOn(site.pages, 'isCategoryPage').and.returnValue(false);
// Always return true when checking if current page is Basket Page
spyOn(site.pages, 'isBasketPage').and.returnValue(true);
```

Besides, it is also needed to see what is passed to the function that will handle the saving data to local storage. Therefore, ‘spyOn’ will be used to help retrieve input in order to compare with the predefined values. It is, however, important to notice that with ‘spyOn’ function, the actual implementation will be replaced by Jasmine’s own implementation of the function. In this case, the function that handles the saving data to local storage has to be executed to save values for coming test. Hence the ‘callThrough’ method is called.

```
spyOn(easy.store, "set").and.callThrough();
```

It is now needed to trigger DOM ready event manually from helper’s library so that the tests can be executed. DOM ready event means that all the HTML has been received and parsed by the browser to the DOM tree.

```
// Initialize conversion tracking
site.conversionTracking.init();
// Trigger DOM ready event
easy.events.trigger(easy.EVENT_DOM_READY);
```

Now the actually assertions for the tests will be written with the familiar expect function. In these assertions, the tests expect the function that will handle saving data to local storage to be called with defined values. In this case, products in the baskets and total basket value are tested.

```
expect(easy.store.set).toHaveBeenCalledWith('productBasket',
[
    {
        id: 'DZ4408',
        name: 'DZ4408 Erkek Kol Saati ',
        category: 'Erkek',
        price: '1199.00',
        quantity: 1 },
    {
        id: 'DZ4405',
        name: 'DZ4405 Erkek Kol Saati',
        category: 'Erkek',
        price: '1319.00',
        quantity: 1
    }
]
);
expect(easy.store.set).toHaveBeenCalledWith('transactionTotal', '1838.50');
```

The second test will verify if after users make the successful order, the product information will be sent to the backend. In order for the test to run correctly there are several conditions that need to be met:

- Current page is a successful payment page, not any other pages.
- The function that handle sending data to the backend needs to by spied on with 'spyOn' function from Jasmine.

In the first condition, similarly to the previous implementation, the 'spyOn' will be applied to methods in page identifier and will return a defined value instead of actual ones.

```
// Always return false when checking if current page is Product Page
spyOn(site.pages, 'isProductPage').and.returnValue(false);
// Always return false when checking if current page is Category Page
spyOn(site.pages, 'isCategoryPage').and.returnValue(false);
// Always return false when checking if current page is Basket Page
spyOn(site.pages, 'isBasketPage').and.returnValue(true);
// Always return true when checking if current page is successful Check-
out Page
spyOn(site.pages, 'isCheckoutPage').and.returnValue(true);
```

Now it is time for the actually assertions for this test. In this assertion, the test verifies if the function handle sending data to backend to be called with defined values. In this case, the products that were saved to the basket in previous test.

```
site.conversionTracking.init();
expectedResult = {
    transactionId: '100002218',
    transactionProducts: [
        {
            id: 'DZ4408',
            name: 'DZ4408 Erkek Kol Saati ',
            category: 'Erkek',
            price: '1199.00',
            quantity: 1 },
        {
            id: 'DZ4405',
            name: 'DZ4405 Erkek Kol Saati',
            category: 'Erkek',
            price: '1319.00',
            quantity: 1
        } ],
    transactionTotal: '1838.50'
};
expect(easy.dataLayer.handleItem).toHaveBeenCalledWith(expectedResult);
```

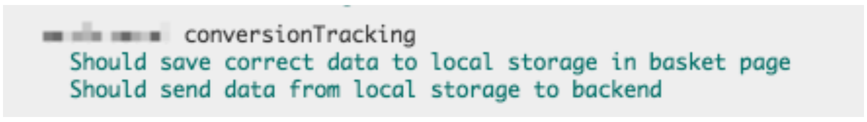
After taking a look at each steps, it is time to assembly all the components into the Jasmine's test code as below

```
describe("customer1 conversionTracking", function () {

    var easy, site, utils, page, oldSiteValue,expectedResult;

    easy = frosmo.easy;
    site = frosmo.site = frosmo.sites.customer1;
    utils = frosmo.sites._utils;
    page = window.jQuery('<div></div>').html(window.fixtures.customer1_basket_page)[0];
    successPage = $('<div></div>').html(window.fixtures.customer1_success_payment_page)[0];

    it("Should save correct data to local storage in basket page", function (){
        spyOn(site.pages, 'isProductPage').and.returnValue(false);
        spyOn(site.pages, 'isCategoryPage').and.returnValue(false);
        spyOn(site.pages, 'isBasketPage').and.returnValue(true);
        spyOn(site.conversionTracking, 'getDom').and.returnValue(page);
        spyOn(easy.store, "set").and.callThrough();
        site.conversionTracking.init();
        easy.events.trigger(easy.EVENT_DOM_READY);
        expect(easy.store.set).toHaveBeenCalledWith('productBasket',
            [
                {
                    id: 'DZ4408',
                    name: 'DZ4408 Erkek Kol Saati ',
                    category: 'Erkek',
                    price: '1199.00',
                    quantity: 1 },
                {
                    id: 'DZ4405',
                    name: 'DZ4405 Erkek Kol Saati',
                    category: 'Erkek',
                    price: '1319.00',
                    quantity: 1
                }
            ]
        );
        expect(easy.store.set).toHaveBeenCalledWith('transactionTotal', '1838.50');
        expect(easy.store.set).toHaveBeenCalledWith('addToCartProducts', [ ]);
    });

    it("Should send data from local storage to backend", function (){
        spyOn(site.conversionTracking, 'getDom').and.returnValue(successPage);
        spyOn(site.pages, 'isProductPage').and.returnValue(false);
        spyOn(site.pages, 'isCategoryPage').and.returnValue(false);
        spyOn(site.pages, 'isBasketPage').and.returnValue(false);
        spyOn(site.pages, 'isCheckoutPage').and.returnValue(true);
        spyOn(easy.dataLayer, "handleItem");
        site.conversionTracking.init();
        expectedResult = {
            transactionId: '100002218',
            transactionProducts: [
                {
                    id: 'DZ4408',
                    name: 'DZ4408 Erkek Kol Saati ',
                    category: 'Erkek',
                    price: '1199.00',
                    quantity: 1 },
                {
                    id: 'DZ4405',
                    name: 'DZ4405 Erkek Kol Saati',
                    category: 'Erkek',
                    price: '1319.00',
                    quantity: 1
                } ],
            transactionTotal: '1838.50'
        };
        expect(easy.dataLayer.handleItem).toHaveBeenCalledWith(expectedResult);
    });


});
```

The test will be run the again with the command

./jasmine.sh customer customer1

After running, Jasmine will run the test on the browser. By opening the result address provided by Jasmine, the above test result can be viewed. The test returns in green colour implying that they have been passed correctly as seen in figure 9.



```
[redacted] conversionTracking
  Should save correct data to local storage in basket page
  Should send data from local storage to backend
```

Figure 9. Expected test result from conversion tracking

6 Discussion

Product tracking unit test includes tests to different attributes of the product in the page. By using loop function, the test helps to reduce a considerable amount of time by avoiding repeated codes. The results in browsers show all in green colour. This helps to indicate that the tests have been implemented correctly and the code works as supposed. Conversion tracking unit test includes tests to verify if the products have been saved correctly in basket page to local storage and sent to the back end from local storage if user has paid the order successfully. After the tests are run, the browser returns green results to both of the tests. This helps to verify that the codes are implemented correctly.

Unit test plays a more and more important role in e-commerce development. There are many methods to cover including test driven development and behaviour development. During the implementations of the tests for this project, Jasmine as a behaviour driven development method was used as a main tool for unit testing.

After implementing the tests, there have been some important things that developers need to understand before or even after the tests. First of all, all the methods that need to be tested by Jasmine have to be exposed to public scope. If the methods are kept in private scope, Jasmine cannot inject its own implementation causing the tests to fail. One example could be the method to get the current URL. Since Jasmine runs locally, the current URL will be the local address instead of the expected URL in the implementation which will cause the code to malfunction in some cases. Besides, Normal native API from browsers such as ‘window’ or ‘document’ should be written insides functions because similarly to the case above, since Jasmine runs on its own local server, the data gotten from browsers’ API will not be the same as it is in production for customers. It is needed to wrap them insides methods such as ‘getWindow’ or ‘getDom’ so that in testing environment, Jasmine can replace those API with its own implementation and return defined values instead. The ‘getWindow’ and ‘getDom’ can be re-written as bellow and should be exposed to global scope: [3.]

```
function getWindow(){
    return window;
}
function getDom(){
    return document;
}
```

In the testing code base, there are functions called insides functions when Jasmine tests are running. In that case, those called functions will run in its own scopes that might lead to unexpected results [2]. In order to fix that, it is important to know how to use function all with proper scopes. Luckily browsers have provided methods such as ‘apply’ or ‘bind’. These methods will execute the codes with the given scopes. [23.]

However, in some browsers such as PhantomJS or old browsers, these methods do not exist when running tests resulting tests to fail. In such cases, the alternative way is to write a pollyfill method for function prototypes as bellow. These help to make sure that the tests will not fail when running in such browsers. [24.]

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function (oThis) {
        var myArgs = Array.prototype.slice.call(arguments, 1);
        var functionToBind = this;
        var fNOP = function () {};
        var fBound = function () {
            return functionToBind.apply(this in-
stanceof fNOP && oThis ? this : oThis, myArgs.concat(Array.proto-
type.slice.call(arguments)));
        };
        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP();
        return fBound;
    };
}
```

Since the code module can be very complicated, it is difficult to actually find out what to test. In this project, the unit tests are written for the key features of the code base. In the conversion tracking unit test, for example, there are tests to check if the data is saved correctly and sent correctly to the database. Small tests will help to cover all necessary cases.

The time to implement these tests varies from developers to developers depending on the experiences. During the scope of this project, there is some feedback retrieved from developers. For new developers, this process can last longer due to the lack of understanding how the system works to write the tests. Besides, they are not aware of what piece of code should be tested. For senior developers, they also often forget to update the tests after changing the codes due to some extra requirements. This results in a considerable extra amount of time to implement a project. However, this helps to keep the codebase consistent and easier to maintain. For some developers, when they are updating some part of the codebase, they are not aware that they are breaking a function somewhere else. This can easily cause errors. If being ignored, developers may deploy it into production that might cause bad consequences.

7 Conclusion

The goal of this project was to understand how unit testing would be used in e-commerce development and its practicalities with Jasmine as a test framework. Writing the test cases requires deep understanding on how the system works and necessary functionalities provided by Jasmine. Mastering Jasmine is a long process, yet it is worth it. The clean documentation and examples provides by Jasmine online document [3] will help developers to grasp basic ideas in a short time with small efforts.

Unit testing helps to increase the quality of the code in a long term. However, it will take time to fully adopt in the design and development of e-commerce development since it is different from tradition software development. This method also requires willingness of developers to learn new testing frameworks as well as understanding the codebase. Besides, it is important to receive support from project managers as well as the management teams as this method will make project delivering time delayed longer than usual due to high learning curve but this will reduce by time as developers will get hands on more and more with it.

References

1 Beck, Kent, Test-driven development by example. Addison-Wesley, 2003.

2 David Flanagan, JavaScript: The definitive guide, 4th edition. O’Reilly Media, 2001

3 jasmine.github.io/. Documentation [Online]

URL: http://jasmine.github.io/

Accessed 5th September, 2016

4 Project Management with the IBM Rational Unified Process: Lessons from the trenches. IBM Press, 2006.

5 Research.IBM.com. Documentation [Online]

URL: http://www.research.ibm.com/haifa/Workshops/verification2007/present/Dor_Nir_web.pdf.

Accessed 5th September, 2016.

6 www.iso.org. Documentation [Online]

URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39064

Accessed: 5th September, 2016.

7 quora.com. Forum [Online]

URL: https://www.quora.com/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development

Accessed: 5th November, 2016.

8 Kent Back, Test Driven Development: By Example Addison-Wesley Longman, 2002.

9 dannorth.net. Blog [Online]

URL: https://dannorth.net/introducing-bdd/

Accessed: 5th November, 2016.

10 www.codemag.com. Documentation [Online]

URL: http://www.codemag.com/article/0805061

Accessed: 5th September, 2016

11 Munish Sethi, Jasmine Cookbook. Packt Publishing, April 2015.

12 Frosmo Website. Documentation [Online]

URL: www.frosmo.com/en/solutions/tech

Accessed: 5th September, 2016

13 Flanagan, David, JavaScript: The Definitive Guide (6th ed.). O'Reilly & Associates, 2011.

14 inc42.com. Documentation [Online]

URL: https://inc42.com/resources/effective-personalisation-website/

Accessed: 5th November, 2016

15 compass.com. Article [Online]

URL: http://blog.compass.co/ecommerce-conversion-rates-benchmarks-2016/

Accessed: 5th November, 2016

16 support.google.com. Documentation [Online]

URL: https://support.google.com/adwords/answer/1722022?hl=en

Accessed: 5th November, 2016

17 Hoek, J., Gendall, P. and Esslemont, D., Market segmentation: A search for the Holy Grail. Journal of Marketing Practice Applied Marketing Science, Vol. 2, no. 1, 1996.

18 www.intelligentcontentconference.com. Article [Online]

URL: http://www.intelligentcontentconference.com/5-ws-adaptive-content/

Accessed: 5th November, 2016

19 www.wired.com. Article [Online]

URL: https://www.wired.com/2012/05/test-everything/

Accessed: 5th November, 2016

20 sublimetext.com. Documentation [Online]

URL: https://www.sublimetext.com/

21 Weimer, J, Research Techniques in Human Engineering. Englewood Cliffs, 1995.

22 api.jquery.com . Documentation [Online]

URL: http://api.jquery.com/html/

Accessed: 5th September, 2016

23 developer.mozilla.org . Documentation [Online]

URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Glo-bal_Objects/Function/apply

Accessed: 5th November, 2016

24 github.com/ariya/phantomjs . Forum [Online]

URL: https://github.com/ariya/phantomjs/issues/10522

Accessed: 5th September, 2016