jamk.fi

Performance testing

Performance measurement of web application

Matus Kapralik

Bachelor's thesis
May 2015

Software Engineering

School of Technology, Communication and Transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s)	Type of publication	Date
Kapralik, Matus	Bachelor's thesis	29.05.2015
		Language of publication: English
	Number of pages 45	Permission for web publication: x

Title of publication
Performance testing
Performance measurement of web application

Double degree programme
ITPRO

Tutor(s)
Salmikangas, Esa

Assigned by
Descom Oyj

Abstract

The main objective of this project was to design and implement a performance test which can provide relevant information for stakeholders. The test was designed for testing the performance of newly developed store which can be improved in the future. An implementation of the new store was based on IBM WebSphere Commerce that is a software platform for cross-channel commerce. E-commerce is a general concept of covering business transaction between organizations of various types.

The implementation of designed test is a process which depends on used software tools. The test was created with an open source application Apache JMeter which is able to measure performance of the web application. One of the main goals of this thesis was to design a testing scenario which is able to simulate a flow of customer behavior on the store web pages. Subsequently implement the designed test via Apache JMeter, execute it and provide desired data.

Results of the thesis are managed by Jenkins CI that is an integration server which accelerates the software development process through automation. In the future the test can be extended with parts that will be considered for testing the performance.

Keywords/tags
Performance testing, web, JMeter, IBM Websphere

Miscellaneous

Contents

Figures

Tables

Acronyms and abbreviations

B2C	business-to-customer
B2G	business-to-government
C2C	customer-to-customer
CPU	Central Processing Unit
DNS	Domain Name System
E-commerce	Electronic commerce
EJB	Enterprise JavaBeans
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/output
Java EE	Java Enterprise Edition
JSP	JavaServer Pages
RAM	Random-access memory
SQL	Structured Query Language

1 Introduction

Today most services have become a part of web environment. In the past, the internet did not cover such a huge amount of population as presently. It is possible to process nearly every kind of service easily via an internet browser. Customers increasingly get used to finding their requests over the internet. At present every seller tries to extend his business to this sphere. The outcome of this compromise is called e-commerce.

E-commerce - commonly known as electronic commerce - refers to buying and selling of goods over the internet. Nowadays, a web transaction increases in many countries proportionally, which causes an enhancement of internet services. This form of trading usually involves transportation of physical items. It is also referred to as business-to-customer, or B2C. There are other known forms such as customer-to-customer (C2C), business-to-business (B2C), and business-to-government (B2G). (Java EE & Java Web Learning, 2015)

Because of increasing the e-commerce all businesses or sellers want to produce the best offer that should increase their revenues. There are many methods which can improve their web solution. For example, a user-friendly design, search engine optimization, advertising etc. Also what is really important is the ability to keep customers for future purchases. A web solution for e-commerce is usually a complex process which consists of requests and responses handled by a server. One of the key improvements which increases shop value is performance. Usually a basic action such as adding a product to shopping cart covers a comprehensive functionality. Developers try to create the best solution that uses optional resources, and the response time is as quick as possible.

In Descom that is a company which provides an ecommerce solutions for customers, testing the performance of an application is done for almost every project with the aim of comparing performance during development process and after finalizing.

The way how to measure the performance of an application is called performance testing. It is a process that sends requests over the internet protocol, receiving

responses and evaluating focused parameters. Testing can be based on testing response time, controlling response text, also exploiting software bugs etc.

This thesis focuses on web performance testing where the objectives can be divided into following subtasks:

- purpose of software testing,
- description of e-commerce application,
- performance testing regarding web application,
- implementation of web performance testing and representation of results.

2 Software testing

2.1 Overview

Software testing is a process of executing an application with the intent of finding software bugs. Testing can also provide information about system quality or an independent view of software to understand risks of software implementation. (Software testing, 2015)

There are two different types of activity, testing and debugging, which are often mistaken with each other. Debugging is a process when developers go through application and try to identify the cause of bugs or defects in code and undertake corrections. Testing, on the other hand, is a systematic exploration of system components with main aim of uncovering and reporting defects. The main difference between testing and debugging is that testing process does not include correction of defects – these are passed on the developer to correct. Therefore, both activities are needed to achieve a quantity result. (Hambling, Morgan, Samaroo, Thompson, & Williams, 2007)

2.2 Static and dynamic testing

2.2.1 Static testing

Test cases are developed using various test techniques for achieving more effective results. Each tester should consider, which method or test techniques is best choice for developed system. Static testing is a method which is used while the code is not yet performed. Failures of designed systems that are tested in a static test are often caused by a human error, namely a mistake in a document such as a specification. Errors are much cheaper to fix than defects or failures. Because of that testing should start as early as possible. Static testing involves techniques such as reviews, which can be effective in preventing defects, e.g. be removing ambiguities and errors from specification documents. Another static technique is known as static analysis which

focuses on structural defects or systematic programming weaknesses that may lead to defects. (Hambling, Morgan, Samaroo, Thompson, & Williams, 2007)

Review technique is exercised manually, whereas static analysis is usually performed automatically using various tools.

A review technique is a systematic examination which is realized by one on more people who try to find and remove errors. Giving a draft document to a colleague to read is the simplest example of a review. This attempt can provide uncovering of errors which can be easily resolved. (Ibid)

Reviews can be used to test everything that is written or typed; this can include documents such as requirement specifications, code, system design, test plans and test cases. The practice of testing specification documents by reviewing them early on in the life cycle helps to identify defects before code implementation which can significantly save resources such as money and time. If the same defects are found in dynamic testing which is performed on a running system, extra cost of initially creating and testing the defective code, diagnosing the source of defect, correcting the problem and rewriting the code to eliminate the defect would incur. (Ibid)

Like reviews, **static analysis** is a testing technique which finds defects before executing the code. The difference between review and static analysis is that static analysis is carried out on written code. The main aim is to find a defect in application source code and software models. A software model is an image of the final solution which is developed by using techniques such as Unified Modeling Language; it is usually generated by the software designer. (Ibid)

Static analysis can find defects that are hard to find during the test execution by analyzing the program code e.g. instructions to the computer can be in the form of control flow graphs and data flows. (Ibid)

2.2.2 Dynamic testing

Dynamic testing is the kind of testing that exercises the program under test with some data. This method involves working with tools where requests are given with inputs and results of responses are checked and compered with the expected values. (Hambling, Morgan, Samaroo, Thompson, & Williams, 2007) Dynamic testing is performed on software that is compiled and executed with parameters such as memory usage. (Dynamic Testing, 2015)

Dynamic testing can be divided into four levels:

- unit testing,
- integration testing,
- system testing,
- acceptance testing.

Unit testing is a method which can test an individual unit of application. In unit testing, a unit can be named as a small testable part of an application. In procedural programming, a unit can be an entire module. In object orientated programming, a unit is often an entire interface such as class; however, it can also be an individual method. Unit testing is usually formed as small code fragment. (Unit testing, 2015) Typically this code can have one or more inputs and often a single output. Unit testing uses white-box testing method which will be more detailed in the following section. (Unit testing, 2015)

Integration testing is the second level of dynamic testing. It occurs after unit testing where units are combined into a group and tested together. The main aim of integration testing is to test the interaction between units and find faults. Integration testing can be divided into testing types by approaches (Integration testing, 2015):

- Big bang is a type of integration testing where all or major units are tested together. This test is aimed at testing whole functionality in a

bundle. The advantage of this approach is saving time in integration testing.

- Top down is used when top level units of software are tested first and lower level units are tested after that.
- Bottom up is almost the same as top down approach but first lower level units are tested and top level units later.
- Sandwich testing approach is a combination of top down and bottom up testing.

System testing is testing based on an integrated system where the purpose is to evaluate the system´s compliance with specific parameters. The condition for system testing is to pass integration testing. This approach of testing falls within the scope of black-box testing. (System testing, 2015)

Acceptance testing is a part of dynamic testing where it is determined if specified business requirements are acceptable for delivery. (Integration testing, 2015)

2.3 The box approach

2.3.1 White box

White box testing (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure is known to the tester. The tester chooses inputs for a known implementation to exercise paths through code and determine appropriate outputs. White box testing can be used on the previous testing methods i.e. unit, integration and system testing. (White box testing, 2015)

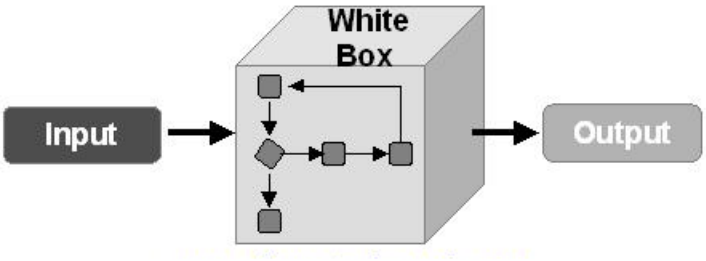


Figure 1 White box testing

White box testing (Figure 1) is a testing at the level of source code. Because of that the tester has to have a deep understanding of code. This method is named so because from the point view of a tester it is like a transparent box inside which one clearly sees. (White box testing, 2015)

2.3.2 Black box

On the other hand, black box is known as Behavioral Testing, where the internal structure or implementation of software is unknown for the tester. These tests can be functional or non-functional.

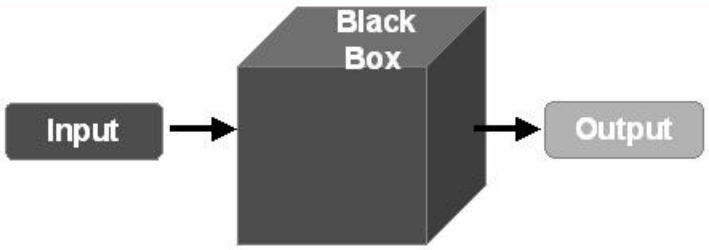


Figure 2 Black box testing

Black box (Figure 2) means that from the point view of the tester it is not possible to look inside. In that case tester does not know, what the implementation on system is. The test is performed by inputs where the system produces outputs which are compared to the required values.

2.3.3 Gray box

Gray box is a testing method which is a combination of both box methods – white and black. In the gray box, the internal structure of a system is partially known. This involves having access to internal data structures and algorithms for purposes of designing the test cases, however, testing at the user, or black-box level.

3 IBM WebSphere Commerce

3.1 Introduction into e-commerce

E-commerce or electronic commerce is defined as buying and selling of products and services where transactions are processed over an electronic network, mainly the internet. Well-known forms of e-commerce forms include (Java EE & Java Web Learning, 2015):

- C2C where transaction takes place between individuals, usually through third-party side such as online auction. A typical example of C2C e-commerce is eBay.
- B2C indicates transactions between businesses and customer.
- B2B is known as trading between businesses and manufacturer.
- B2G is trading between businesses and government.

3.2 IBM WebSphere Commerce

IBM WebSphere Commerce provides a powerful cross-channel commerce platform that can be used by companies of all sizes, small businesses, large enterprises and many different businesses. It is a system where business user can produce and manage precision marketing campaigns, promotions, catalogs and merchandising across all sales channels. IBM WebSphere Commerce can be named also omni-channel e-commerce platform that enables business-to-consumer and business-to-business sales to customers across channels. (WebSphere Commerce product overview, 2015)

3.2.1 Architecture

The following figure (Figure 3) is a simplified view of software components that are related with WebSphere Commerce.

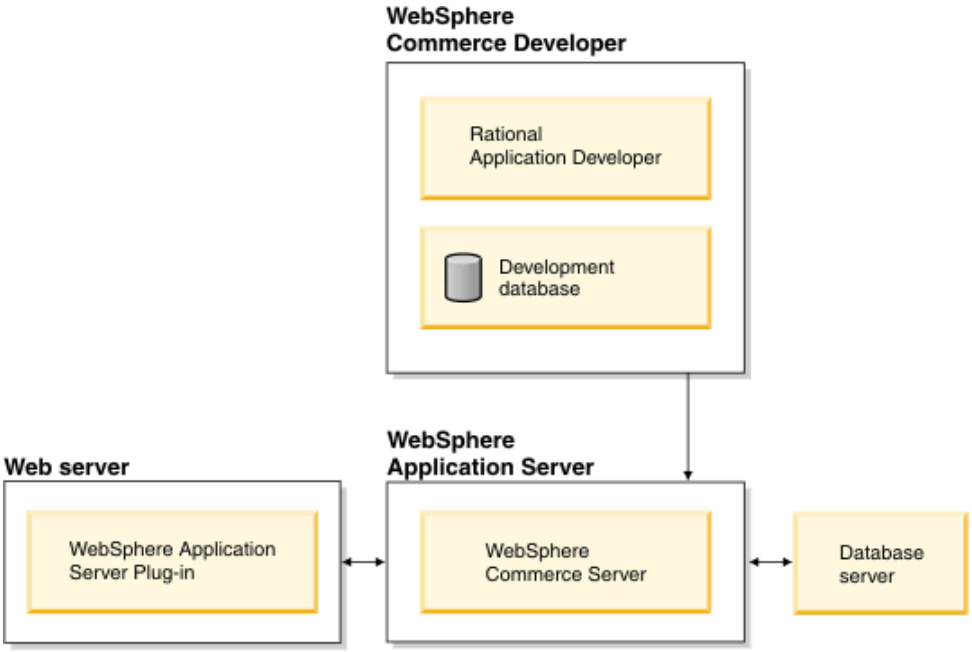


Figure 3 IBM WebSphere Software components

The first contact with WebSphere Commerce comes through Web server as an HTTP request. The connection between Web server and WebSphere application server is ensured via WebSphere Application Server Plug-in. The database server holds most of application data which include products and customer data. (WebSphere Commerce common architecture, 2015)

Developers use Rational Application Developer to perform the following tasks:

- create and customize storefront assets such as JSP and HTML pages,
- create and modify business logic in Java,
- create and modify access beans and EJB entity beans,
- test code and storefront assets,
- create and modify Web services.

3.2.2 Application layer

The following diagram (Figure 4) shows application layers that compose the application architecture:

Figure 4 WebSphere Commerce application layers

Business models describe a scenario which various parties use to achieve their needs. Business models which are provided by WebSphere commerce are:

- B2B direct,
- consumer direct,
- demand chain,
- hosting,
- supply chain.

Business processes are represented by processes which are available in WebSphere Commerce divided by the business model. (WebSphere Commerce application layers, 2015)

Presentation layer is responsible for displaying results. There are two types for presenting results - web and rich client. For web presentation the display is rendering JSP files, for client the presentation is rendered with Eclipse view. (Ibid)

Service layer is a mechanism that can access WebSphere Commerce business logic. Two mechanism are supported (WebSphere Commerce application layers, 2015):

- local Java binding,
- web services.

Business logic is presented by a command pattern. There are two types of implemented commands (WebSphere Commerce application layers, 2015):

- controller commands - accessible by the presentation layer and used as a coordinator of tasks,
- task commands - not accessible by the presentation layer but called from the controller commands. This command type is used to implement business rules.

Persistence layer is used for recording data and operations of the WebSphere Commerce. (WebSphere Commerce application layers, 2015)

Database schema is designed especially for WebSphere Commerce which includes over 600 tables and covers required data. The schema supports persistence requirements for WebSphere Commerce subsystems such as Order, Catalog, Member, Marketing, Trading etc. WebSphere commerce supports both DB2 and Oracle relation databases. (WebSphere Commerce application layers, 2015)

4 Web application performance testing

4.1 Introduction to web performance testing

The main aim of Web Performance Testing is to measure the actual performance of a web application and evaluate performance that the application could provide, identifying, moreover, possible bottlenecks and providing useful advice for fixing the problem (tuning of hardware components, modification of software or tuning system parameters). (Cassone, Elia, Gotta, Mola, & Pinnola)

Before start to measure web performance effectively quite important is to know the architecture (Figure 5). This architecture includes (Ibid) :

- web browser as software client where an application runs,
- internet service providers that affect internet access,
- web servers which are applications that are able to meet requests from client (browser). Subsequently requests are forwarded to a web application which can be on the same machine or on another server,
- application server is a place where the code of the application runs. Requests come there from web servers where are handled and provide responses,
- database persists data of the application. Accessing the data can be difficult and because of that the access time can be too high.

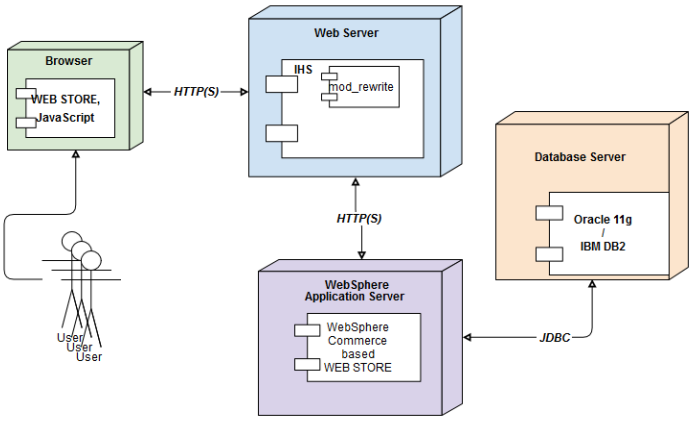


Figure 5 Web application architecture

4.2 Activities of performance testing

The following section discusses seven activities (Figure 6) that most commonly occur across successful performance testing.

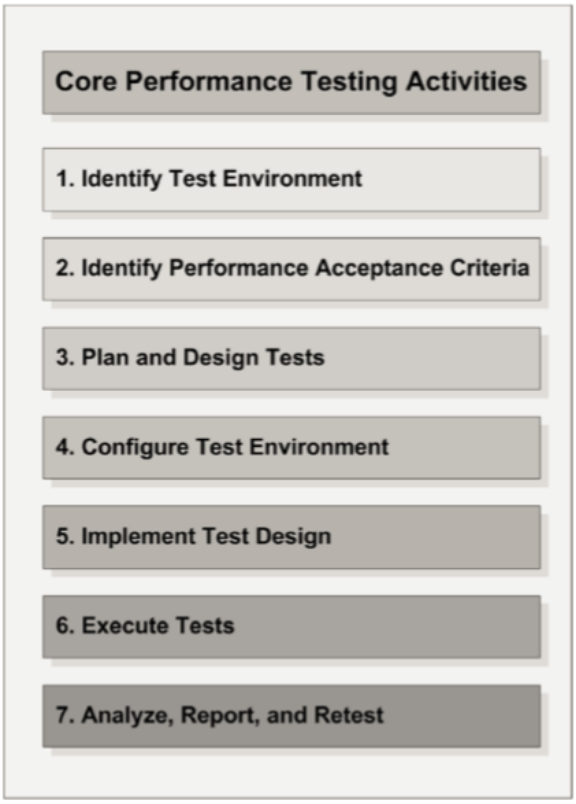


Figure 6 Activities of performance testing

4.2.1 Identify Test Environment

The aim of system evaluation is to gather information about the whole project, the functions of the system, the expected user activities, the system architecture and other details which can be conducive for performance testing to achieve specific needs of the project. This information offers a fundamentals for collecting performance requirements and goals, workload characterizing, creating strategies and plans for performance testing. This process provides a base for determining acceptable performance; defining performance requirements of the software or components; identifying any risks to the effort before testing even begins. (Meier, Farre, Bansode, Barber, & Rea, 2007)

An environment where performance tests are realized with all tools and hardware represents test environment. There is also important to mention that it is not only the software and physical environment that influence performance testing; however, objectives of test itself too. Some critical factors to consider are listed below (Ibid):

- hardware – configuration, machine hardware (processor, RAM, etc.),
- network – network architecture, load balancing, cluster and DNS configurations,
- software – other software installed in environment, logging levels,
- external factors – interaction with other systems, volume and type of additional traffic, scheduled processes, updates.

4.2.2 Identify Performance Acceptance Criteria

The effort of performance testing is to determine the objectives for identifying potential risks, changes and opportunities for improvements. That means start or at least estimate identifying desired characteristics in performance testing of the application early in development cycle. Desired characteristics are meant as results that are evaluated and accepted as good performance by users and stakeholders. These characteristics which often correlate to user and stakeholder satisfaction are usually classified into following criteria (Meier, Farre, Bansode, Barber, & Rea, 2007):

- **Response time** which means the time taken from one system node to respond to the request of another. For example, opening and displaying catalog pages should not take more than two seconds.
- **Throughput** is considered as number of units which can be handled per specific time. It could be for example requests per second.
- **Resource utilization** that is the cost of the project in terms of system resources. These resources could be processor, memory, disk I/O, network I/O

4.2.3 Plan and Design Tests

The most common purpose of web performance testing is to simulate user scenarios as realistically as possible. It means to reproduce user behavior while using web

interface. The tested workloads have to present real-world scenarios. For creating a reasonably accurate representation of reality, understanding business context of application which is used is needed. (Meier, Farre, Bansode, Barber, & Rea, 2007)

Design performance testing is usually a difficult process where the tester tries to reproduce user way of application using. The tester should consider behaving of the single user or group, what is the most interesting for them, how long takes action which can be inspecting of products, reading text information, etc. It is really important to design the most common test to reality for simulating processing which can lead to performance improvements. In addition, metrics can help to identify problem areas and bottlenecks in the tested application.

4.2.4 Configure Test Environment

Configuring the test environment, tools, and resources are necessary to achieve the planned scenario. It is important to ensure that the test environment is instrumented for resource monitoring to be helpful with the more efficient analysis of the results. Depending on the company, a separate team can be responsible for setting up the test tools, while another one may be responsible for configuring other aspects such as resource monitoring. In other organizations, a single team can be responsible for setting up all aspects. (Erinle, 2013)

4.2.5 Implement Test Design

Implementation and execution of performance testing is extremely tool specific. Regardless of the chosen tool, creating performance testing usually involves scripting designed scenario and also enhances it combining with others scenarios for producing a complete workload model. There are numerous available tools intended for performance testing, both free and commercial. (Meier, Farre, Bansode, Barber, & Rea, 2007)

4.2.6 Execute Tests

Executing tests is the phase when the implemented test are run. The tests are executed by a tool which was chosen as the best possibility for this purpose. Executing tests

often takes quite enough time needed for reaching reliable results that could be helpful for improving the application performance.

The first step for executing the test is verifying prepared scenario, which can be prepared as a script or other resource. This process is done by a tool that produces results from measuring. If everything is all right, the test is executed and subsequently provides data from testing. It is a good practice to print the process, for example, log files, console output, where the tester can check the progress of testing or inspect ongoing tasks. This can be useful for fixing the test, if something is not working well.

4.2.7 Analyze, Report and Retest

This process is exercised by reviewing each successful test and identifying the bottleneck areas. The bottlenecks can be application, database or system related. System-related bottlenecks may conduct to some infrastructure changes which can increase available memory for application, reduce consumption of CPU, increasing or decreasing thread pool size, reconfiguring network setting etc. Database-related bottlenecks may lead to analyzing database I/O operations, profiling SQL queries, introducing additional indexes, changing table page size and a great deal more. Application related bottlenecks may be conducted to activities such as refactoring application components, reducing memory consumption of application, etc. If the identified bottlenecks are addressed, the test should be executed again and compared with the previous testing. For getting relevant results, the test has to be planned with the same parameters, also it is good to run it in the same time. For example, if you are running one test in the evening and during the day with the same parameters, the results might be influenced by traffic. This process repeats until the performance goals of the project have been met. (Meier, Farre, Bansode, Barber, & Rea, 2007)

4.3 Performance Testing Techniques

4.3.1 Load testing

The one of the most used techniques for measuring performance is called load testing. The main aim of this method is to determine web application’s behavior under normal

and anticipated peak load condition. The process of this testing is based on gradual increasing of resources. It means that a test usually begins to load with a small number of virtual users and the incrementally increases from normal to peak. Using this process it is possible to observe how the application during this gradually increasing load testing performs. For example if the testing purpose is to find the point when the response time is lower than five seconds the resources are incremented while this point has been crossed. (Ibid)

4.3.2 Stress testing

Stress test is a kind of test where the tested environment is exposed under an extreme condition. This test is performed because of determining the application reliability, robustness, availability and identify application problems that arise under extreme conditions. It can includes heavy loads, high concurrency and hardware resources. Proper stress testing is useful in finding synchronization and timing bugs, interlock problems, priority problems, and resource loss bugs. The main aim of exercising stress test to the point of maximum load is to find issues that can be potentially harmful. Stress test usually involves one or more production scenarios which are run in various stress conditions. For example running application on machine where some other application consume considerable amount of resources. (Ibid)

5 Implementation

5.1 Apache JMeter

The previous chapter covered the fundamentals of performance testing. One of the core activities was the implementation of defined testing scenarios which is extremely tool specific, from free to commercial. This solution is provided by free, open source cross platform desktop application Apache JMeter developed by The Apache Software foundation. JMeter allows to simulate various scenarios with multiple concurrent users with goals such as identifying the system bottleneck, resources problem etc. The way of performance testing is using dynamic type of testing with a black box approach. It means that tests are provided on the deployed application, however, the tester does not know, how the code is handled.

This tool simulates the user activity, however, it should not be misguided with browsers. It does not support all browsers operation as executing JavaScript in HTML pages, render HTML pages. It provides opportunity to process requests and responses and then collect aimed results. Also there is a limitation how many users can run on a single machine. It depends on the environment specification and the tested scenario. For example a single machine with 2.2GHz processor and 8GB of RAM can handle about 250-450 users. (Erinle, 2013)

5.2 Web Test Plan

There are many possibilities how to create a scenario for performance testing purposes. It can be testing only a unit, integration or the whole system. This sample tests the usual user activities on web page. The responsible system that operates web and application server is IBM WebSphere Commerce, so the aim of the testing is to find a point where the application becomes fragile, for example when the response time crosses the maximum allowed value, when it provides some errors and so on. The following sample demonstrates the used approach which consists of two parts:

- *Crawling phase* that is used for collecting data and processing scenario too. The process starts at the front page and subsequently continue on the process that depends on the previous steps. Dependency because of extracting words which forms a path for HTTP requests.
- *Execution in random order* that follows the first phase. HTTP requests are now possible to handle in random order, because all needed expressions for building a path of requests are extracted. For example, request for handling specific product does not need open category of product first, however, it can be tested directly.

All domain names are changed in order to keep the company customers unknown.

5.2.1 Preparing test scenario

Apache JMeter provides a bundle of components that can properly handle a specific user scenario. It is a good manner to prepare some variables that are used many times in various components. The tester does not have to use particular values in each component independently, just parametrize them once with the predefined values from *User Defined Variables* component which is shown in Figure 7.

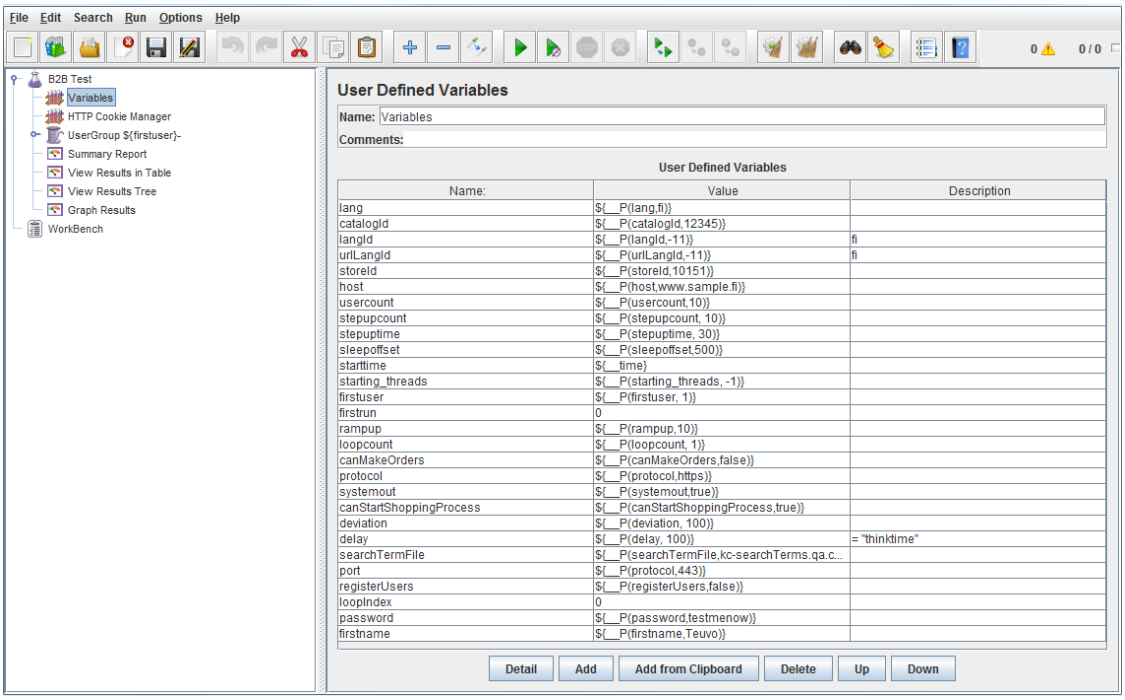


Figure 7 User Defined Variables

In this component there are two important parameters:

- *name* – used as a reference for defined value that can be used in any component,
- *value* – set the value for name. In this field the function *__P(property, default value)* can be used. It returns the value of property, however, if property is not defined, it uses the default value.

If the HTTP request, and the response contains a cookie, *HTTP Cookie Manager* (Figure 8) automatically stores it and uses for the future requests to the particular web site. Every thread has its own cookie storage area.

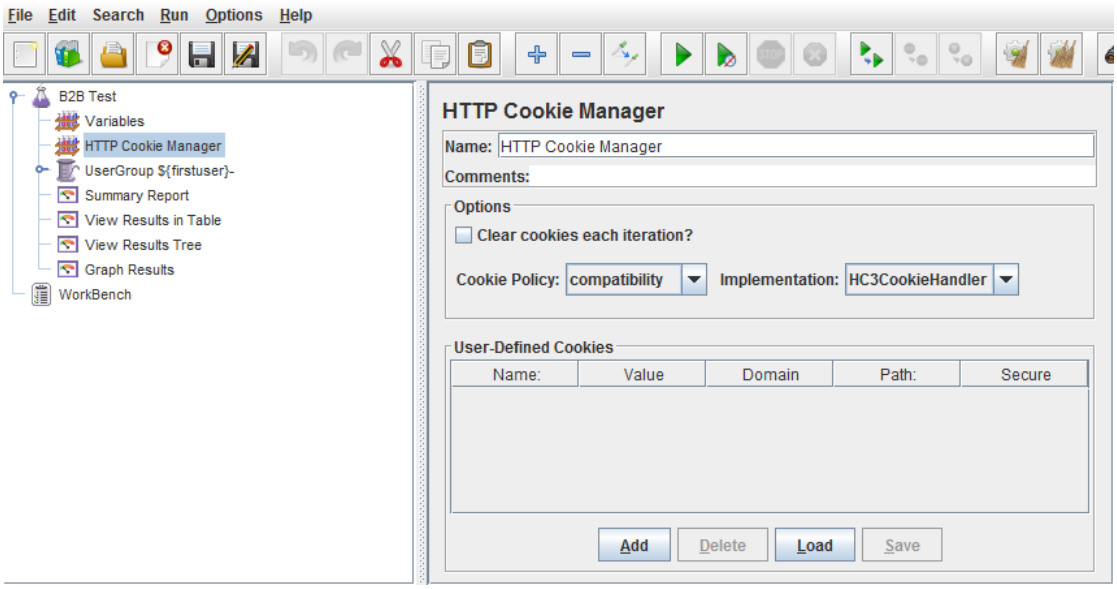


Figure 8 HTTP Cookie Manager

5.2.2 User defined scenario

Every user is represented as a thread. The scenario for every thread is defined in the component called *Thread Group* (Figure 9). There are three thread properties that were set for this scenario:

- *number of threads* that represents number of simulated "users",
- *ramp-up period* that set the time to start all threads,
- *loop count* which is the number of times to performing the test.

All these fields use defined parameters or variables. These three variables are taken from *User Defined Variables* component.

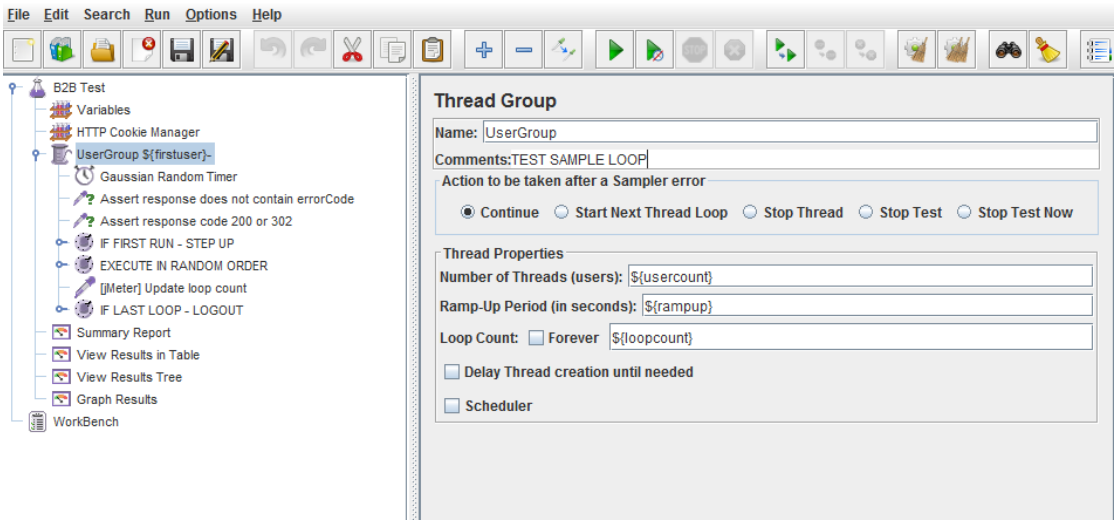


Figure 9 Thread Group

Another component used in this scenario is *Gaussian Random Timer* (Figure 10). If the test should perform a real world scenario it has to behave like a real user. This component causes a delay between particular actions, which means that if a user opens a page on web he/she checks for relevant data for him/her which usually takes a time. How long it takes can be defined by deviation and offset properties. According to the analytics, the average delay between clicks (actions/ navigations) takes 45 seconds.

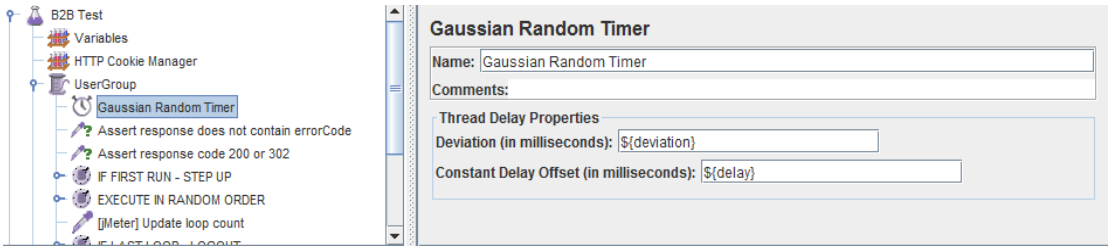


Figure 10 Gaussian Random Timer

To check the response status *Response Assertion* component is used (Figure 11). This component is important because it produces overview about successful responses for sample.

In this case two response assertion controllers are used:

- the first is for handling responses with HTTP code 200 or 302 which means that the request was OK or FOUND. For this test it means successful response.
- The second one is for handling unsuccessful response, however, the implemented system provides only successful responses. Unsuccessful responses show generic error on pages that is handled by searching "errorCode" keyword in response body.

This component provides opportunity to select where the response pattern is checked (text response, response code, response headers etc.).

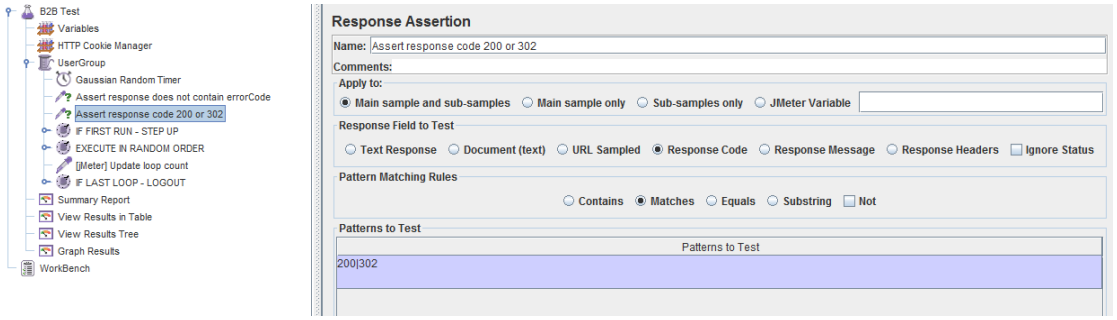


Figure 11 Response Assertion

After preparing all important components for threads the flow of actions can start. This scenario considers that firstly data will be extracted by passing the web as a real user. After that, when all data will be prepared for next step, the test can start to perform executing in random order. The component that is responsible for checking this condition is *If Controller* (Figure 12).

This controller can be considered as a container for other actions. It involves many other components which can be processed only if the condition passes.

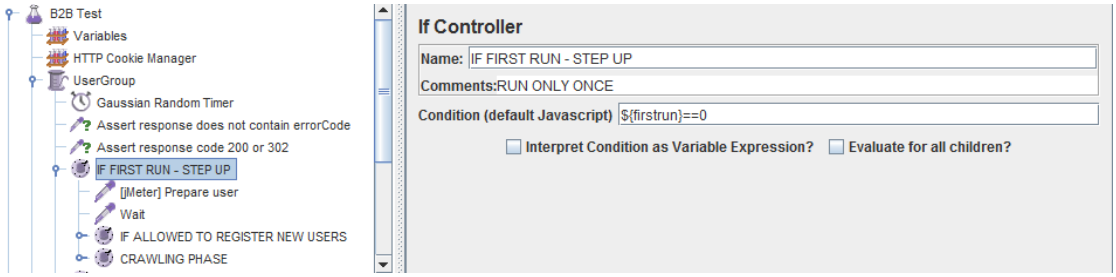


Figure 12 If Controller

The *firstrun* variable is defined in *User Defined Variables* component and this condition which is default JavaScript syntax compares this defined variable with value 0. In this test value 0 means that the test is performed for the first time. If the values are equal the following components in order are performed.

Very useful component that can be used for scripting is a *BeanShell Sampler* (Figure 13). For example, via this sampler the tester can set variables, also use it as log and prints actual progress etc. In actual test step the *firstrun* variable is set to 1 that means the next loop fails this container. Also, it sets the parameter that is intended for registration action and prints timestamp about user and timing.

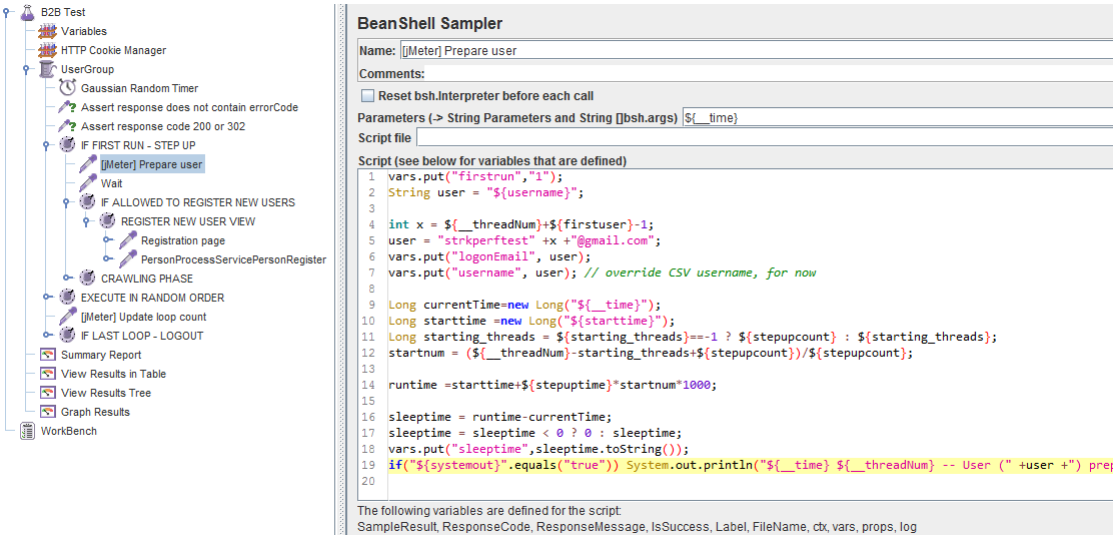


Figure 13 BeanShell Sampler

The test action controller is used for sleeping predefined randomized time to add users to the test gradually (Figure 14). It is a kind of sampler that can do actions such as

pause, stop and move to next iteration. Also, there is a possibility to choose the target for this test action. It can be a single current thread or the whole thread group. The time while the thread is paused is set by the duration parameter.

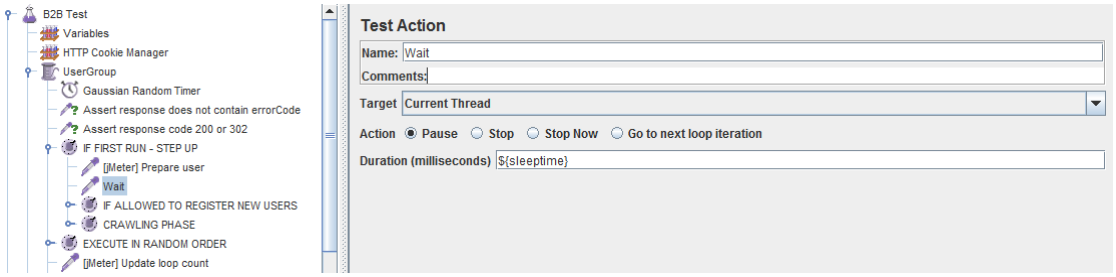


Figure 14 Test Action

The next step of this flow is action for registration new user. It is the same as real user opens web page, chooses registration of new user and fills all required field, however, the first step is to check if the registration action is allowed in test scenario. It can be disabled because if the test was already performed registration, there is not needed to create new users. It fails for sure, because the users with created username already exist in the tested system. *Simple Controller* is used for every action (Figure 15). It is a common container that is good to use for keeping all requests in one logical area.

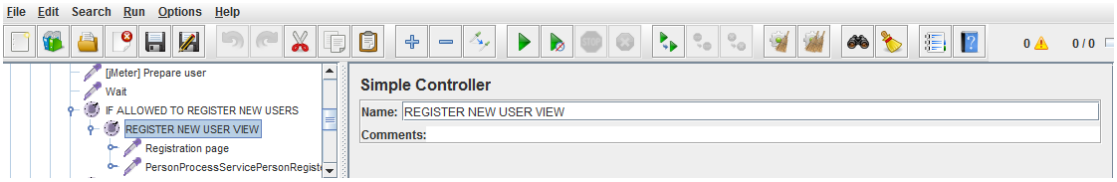


Figure 15 Simple Controller

In this controller a *HTTP Request* is set. This component is responsible for handling request and works with the responses. Basically this component is processing actions that are started after user clicks on any web page action. It can be opening categories, products, adding products to cart etc.

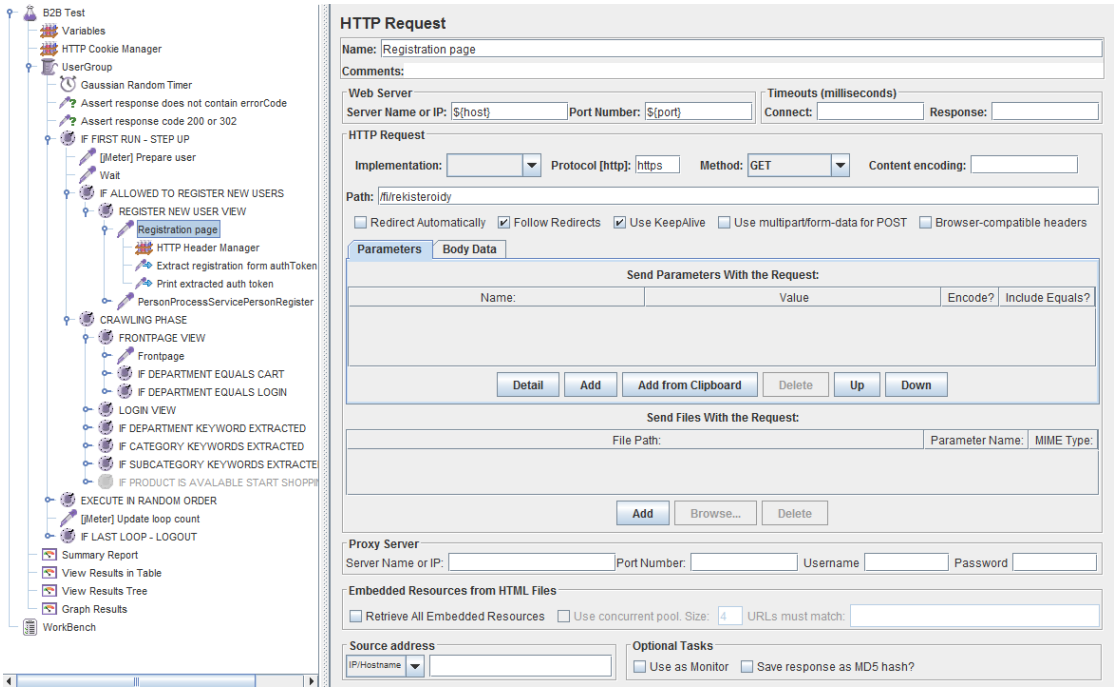


Figure 16 HTTP Request

In this case *HTTP Request* sampler is required to complete a few parameters in configuration view:

- *Web server***,** where the server name is set (comes from defined values). Port is set also, because value 80 is set as default and the server uses secured protocol which is 443.
- *Http request*, where there is a need to set the path of request. This path can be easily reached by some web tools such as Firebug that monitors the activity in the browser. Selected field *Follow Redirects* checks if the response is redirect and follows it. *Use KeepAlive* sets the keep alive header.

HTTP Request can have an additional components that help to pass a request or process another functionality. Every *HTTP request* uses *HTTP Header Manager* (Figure 17) that adds or overrides HTTP request header.

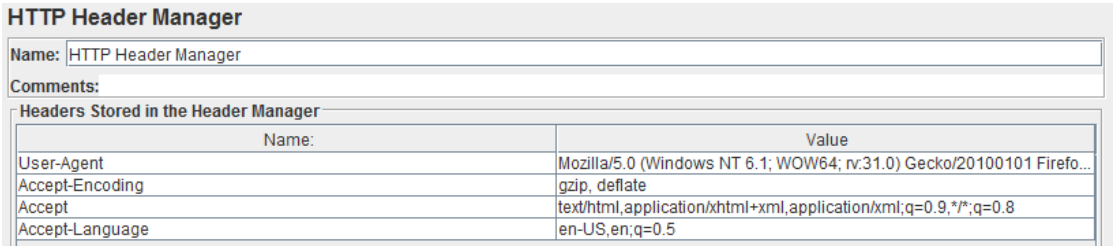


Figure 17 HTTP Header Manager

For retrieving data for the following request is used *Regular Expression Extractor*. All next requests depend on the previous ones because through this component extracted keywords are used in *HTTP request*.

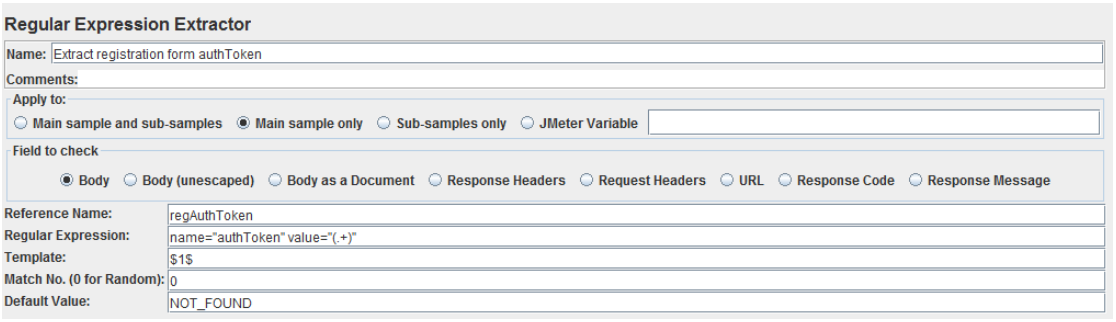


Figure 18 Regular Expression Extractor

The tester chooses for which sample is extracted value applied, select field that is checked and set required following 5 fields:

- *reference name* that can be used in other component as a reference,
- *regular expression* which is expression for parsing response data,
- *template* that selects group from expression that is used (1 refers to group 1),
- *match number* where value 0 means, if there are more results from parsed data take random one,
- *default value* that sets the defined value if there are no matches with regular expression.

A best practice is to print the results while the test is running to be sure that all data are passing right, regular expressions are parsing responses with proper expression etc. A component that is used for this action is called *BeanShell PostProcessor*. It is

the same as *BeanShell Sampler* shown in Figure 13. The difference between these two samplers is in the execution order. *BeanShell S*ampler is executed in order of flow, however, *BeanShell PostProcessor* is executed after a specific request.

The next step after registration is the crawling phase where the scenario starts on the front page and subsequently follows logical flow of user activity on the web.

The front page activity (Figure 19) consists of the following components:

- *HTTP Request,*
- *HTTP Header Manager,*
- *Regular expression Extractor,*
- *BeanShell PostProcessor*.

In this request are extracted multiple values such as department, category and subcategory keyword. Also there are two if controllers, because department keyword can parse also value for cart and login. This keywords have the same structure in response text, therefore they are set to static value if they are selected.

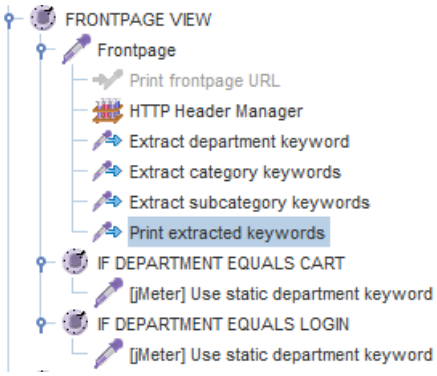


Figure 19 Front page step

The logon page step tests attempt to log in (Figure 20). This step contains two HTTP requests. The first opens the login page and the second one signs up. From the login page authorization token is extracted via regular expression extractor that is needed for sign up. Username and password used in this request were set in *BeanShell*

Sampler (Figure 13). Logon request has its own *Response Assertion* for handling unsuccessful logon.

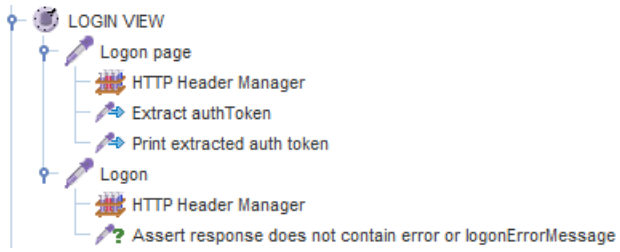


Figure 20 Logon step

After extracting the keyword from the front page the department page is executed the first in order. This action does not extract anything. It is only an HTTP request that simulates the opening of the department page. Category and subcategory are different. In those views the keyword for the product is extracted. In subcategory view also the product availability for shopping action is checked.

For the purpose of check the product availability *While Controller* is used (Figure 21). This controller has same structure as *If Controller*, however, it is performed while the condition is applied.

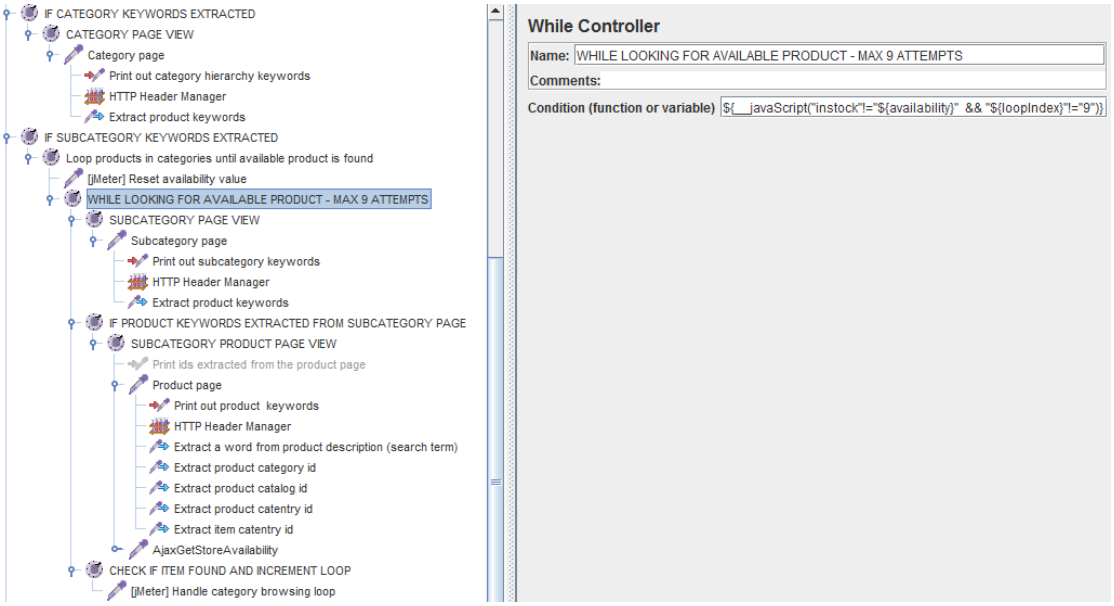


Figure 21 Extracting product keyword

Inside *While Controller* the subcategory view is performing where the products are extracted. Subsequently *If Controller* checks if the product is extracted and starts executing the product page and availability action. After executing these two requests via *BeanShell Sampler* it is checked if the available product was found and the loop value is incremented. The While controller repeats maximum nine times or lower if the available product was found.

If the available product was extracted the shopping process can start (Figure 22). This process consists of controllers that handle adding the extracted product to the shopping cart, user shopping process and removing the order.

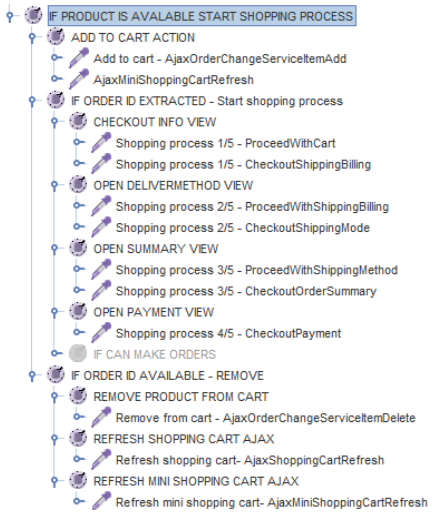


Figure 22 Shopping process

Add to cart action has two requests:

- the first one adds the product to the order and subsequently order id and order item id is extracted via regular expression extractor,
- the second is a simple request for refreshing the mini shopping cart.

If the order id is extracted the shopping process can start. This process is basically set of five actions:

- This process starts with checkout where two requests are measured – ProceedWithCart and CheckoutShippingBilling. From the second mentioned request user data such as name, shipping address are extracted.
- After the checkout view the delivery method view is executed. There are also two similar requests as in the checkout view.
- The third method has the same scenario as the previous two views. All requests that are passing these views were collected via the browser tool that can catch requests and then were used for the shopping process.
- Even this process has five actions, the last one is disabled because it cause a real order that was not possible for this test. Because of that the test ends in this fourth step which opens the payment view.

When the shopping process is over, afterwards the test removes the product from order if it is available (Figure 23). Firstly *If Controller* checks if an order exists and if so the flow of removing the product can start. This action is provided by three requests that remove the product from the order in the system, the shopping cart and the mini shopping cart.

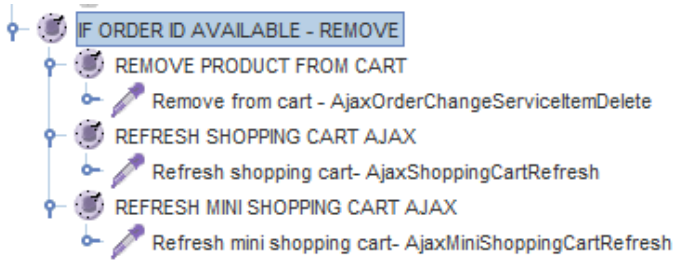


Figure 23 Remove product from order

In the phase of executing in random order the same controllers that are in crawling phase are used, however, the crawling phase was performed mainly because of extracting keywords of this phase. In the thread loop count it was defined how many times a thread is executed. The crawling phase is exercised only in the first loop because of extracting keywords purpose. In executing in random order phase are two additional controllers (Figure 24):

- controller for searching a term,
- controller for opening shopping cart view.

Both of them are simple HTTP requests, however, the controller for searching has *If Controller* that checks if the search term was extracted via regular expression extractor.

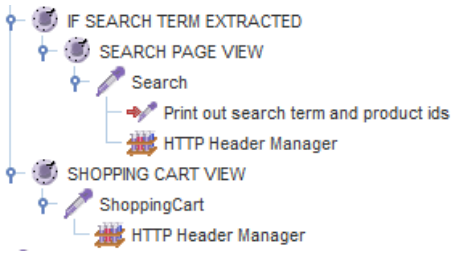


Figure 24 Search and shopping cart controller

The last step of this scenario is logout (Figure 25). For this purpose is used *If Controller* that checks if the count of the loop is equal to the value set in the Thread Group component. If this values are equal the HTTP request for log off is executed.

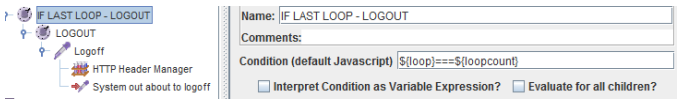
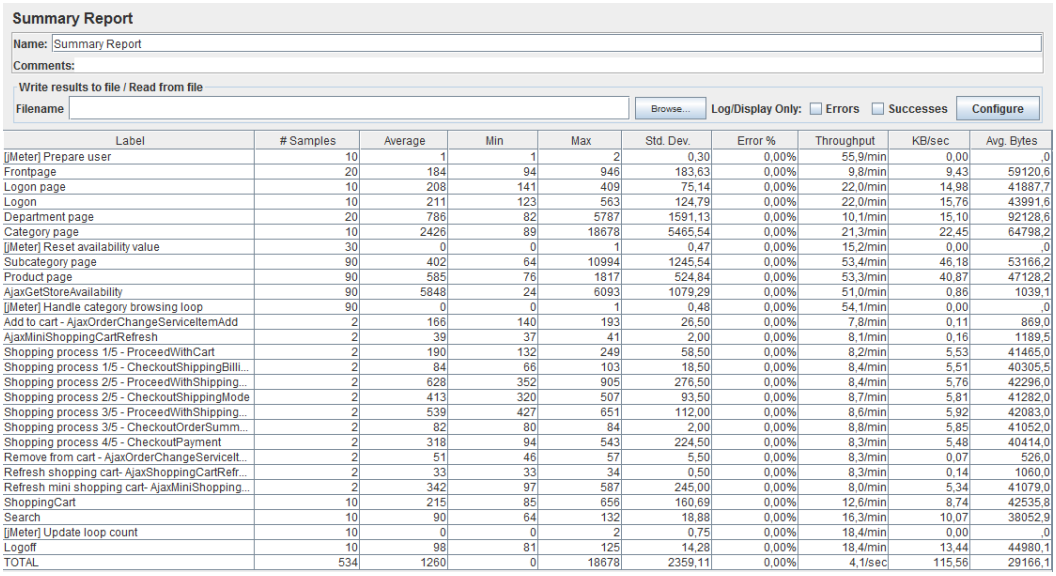


Figure 25 Log off action

6 Results

There are multiple ways how the results from the implemented test can be presented. It can be in the form of table or graphical representation. However, it is important to know, what the numbers of the table or curves in graph mean. Apache JMeter can provide both forms. For this approach these resources are good for checking the progress of the designed test (Figure 26).



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename | Browse... | Log/Display Only: Errors Successes | Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
[jMeter] Prepare user	10	1	1	2	0,30	0,00%	55,9/min	0,00	,0
Frontpage	20	184	94	946	183,63	0,00%	9,8/min	9,43	59120,6
Logon page	10	208	141	409	75,14	0,00%	22,0/min	14,98	41887,7
Logon	10	211	123	563	124,79	0,00%	22,0/min	15,76	43991,6
Department page	20	786	82	5787	1591,13	0,00%	10,1/min	15,10	92128,6
Category page	10	2426	89	18678	5465,54	0,00%	21,3/min	22,45	64798,2
[jMeter] Reset availability value	30	0	0	1	0,47	0,00%	15,2/min	0,00	,0
Subcategory page	90	402	64	10994	1245,54	0,00%	53,4/min	46,18	53166,2
Product page	90	585	76	1817	524,84	0,00%	53,3/min	40,87	47128,2
AjaxGetStoreAvailability	90	5848	24	6093	1079,29	0,00%	51,0/min	0,86	1039,1
[jMeter] Handle category browsing loop	90	0	0	1	0,48	0,00%	54,1/min	0,00	,0
Add to cart - AjaxOrderChangeServiceItemAdd	2	166	140	193	26,50	0,00%	7,8/min	0,11	869,0
AjaxMiniShoppingCartRefresh	2	39	37	41	2,00	0,00%	8,1/min	0,16	1189,5
Shopping process 1/5 - ProceedWithCart	2	190	132	249	58,50	0,00%	8,2/min	5,53	41465,0
Shopping process 1/5 - CheckoutShippingBilli...	2	84	66	103	18,50	0,00%	8,4/min	5,51	40305,5
Shopping process 2/5 - ProceedWithShipping...	2	628	352	905	276,50	0,00%	8,4/min	5,76	42296,0
Shopping process 2/5 - CheckoutShippingMode	2	413	320	507	93,50	0,00%	8,7/min	5,81	41282,0
Shopping process 3/5 - ProceedWithShipping...	2	539	427	651	112,00	0,00%	8,6/min	5,92	42083,0
Shopping process 3/5 - CheckoutOrderSumm...	2	82	80	84	2,00	0,00%	8,8/min	5,85	41052,0
Shopping process 4/5 - CheckoutPayment	2	318	94	543	224,50	0,00%	8,3/min	5,48	40414,0
Remove from cart - AjaxOrderChangeServiceIt...	2	51	46	57	5,50	0,00%	8,3/min	0,07	526,0
Refresh shopping cart- AjaxShoppingCartRefr...	2	33	33	34	0,50	0,00%	8,3/min	0,14	1060,0
Refresh mini shopping cart- AjaxMiniShopping...	2	342	97	587	245,00	0,00%	8,0/min	5,34	41079,0
ShoppingCart	10	215	85	656	160,69	0,00%	12,6/min	8,74	42535,8
Search	10	90	64	132	18,88	0,00%	16,3/min	10,07	38052,9
[jMeter] Update loop count	10	0	0	2	0,75	0,00%	18,4/min	0,00	,0
Logoff	10	98	81	125	14,28	0,00%	18,4/min	13,44	44980,1
TOTAL	534	1260	0	18678	2359,11	0,00%	4,1/sec	115,56	29166,1

Figure 26 Apache JMeter Summary Report

However, the implemented testing scenario was executed in Apache JMeter environment. For this approach Jenkins CI server was used. It is an open-source integration server that accelerates the software development process through automation. This server is able to manage and control the development process such as deployment, build, test, package etc. The following tested sample was executed with parameters shown in Table 1.

Table 1 Testing parameters

Name	Value	Description
Loop count	50	Total loop number of user thread
User count	300	Simulated number of users
Delay	7500ms	Delay value for gauss random timer
Deviation	2500ms	Deviation value for gauss random time
Step up time	60	Value that sleeps thread for predefined time to add users gradually

Results from the sample test that were performed in Jenkins CI server are shown in Table 2.

Table 2 Table of results

Request name	count	average [ms]	median [ms]	90% line	min [ms]	max [ms]	error [%]
Logon page	300	137	107	179	72	3044	0.00
Logon	300	976	692	1693	173	6625	0.01
Frontpage	15300	348	218	786	46	6071	0.00
Department page	15300	453	241	1151	47	5835	0.00
Category page - brand	8683	428	223	1109	43	5520	0.00
ShoppingCart	15300	1094	816	2408	83	7352	0.00
Product page	5428	2316	1776	4316	166	11894	0.00
Add to cart - AjaxOrderChangeServiceItemAdd	53	562	294	1455	79	2336	0.43
Search	5883	619	255	1729	65	6375	0.00
AjaxMiniShoppingCartRefresh	53	724	326	2293	104	3027	0.00
Shopping process 1/5 - ProceedWithCart	26	469	255	1022	102	2803	0.00
Shopping process 1/5 - CheckoutShippingBilling	26	399	214	636	97	1767	0.00
Shopping process 2/5 - ProceedWithShippingBilling	26	377	243	661	99	1544	0.00
Shopping process 2/5 - CheckoutShippingMode	26	530	196	1463	100	2175	0.00
Refresh mini shopping cart- AjaxMiniShoppingCartRefresh	7521	582	342	1513	71	6240	0.00
Shopping process 3/5 - ProceedWithShippingMethod	26	544	251	1588	97	2003	0.00
Shopping process 3/5 - CheckoutOrderSummary	26	404	242	881	97	1563	0.00
Shopping process 4/5 - CheckoutPayment	26	405	193	1140	96	2226	0.00
Remove from cart - AjaxOrderChangeServiceItemDelete	26	136	119	214	40	408	0.00
Refresh shopping cart- AjaxShoppingCartRefresh	26	391	211	828	97	1609	0.00
Logoff	300	239	201	342	105	2874	0.00
TOTAL	94292	665	308	1609	40	17275	0.00

In Table 2 the values that are useful for understanding how long requests taken are displayed. In this result table are also numbers of processing specific request and error rate that means unsuccessful perform of action. This interpretation of table is more

advance because exact numbers of duration, error and threads are shown there. The test produces also the graphical view which is easier for understanding and also explanation for customers. The test provides more graphs such as throughput, latency, number of transactions etc. However, usually for customers the most interesting results are these two graphs (Figure 27, Figure 28).

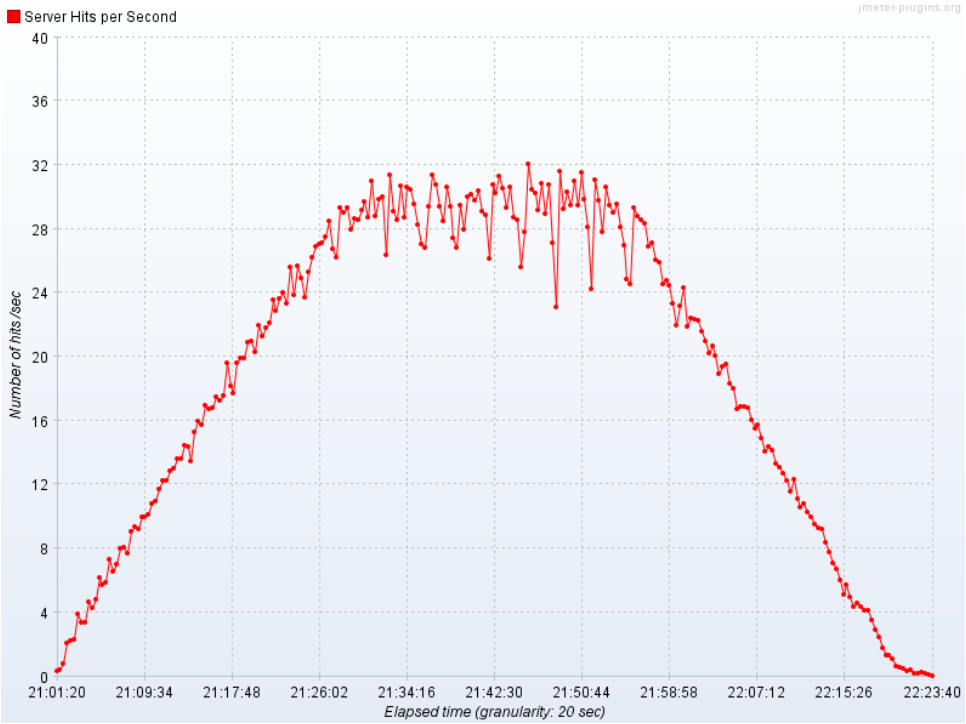


Figure 27 Hits per second

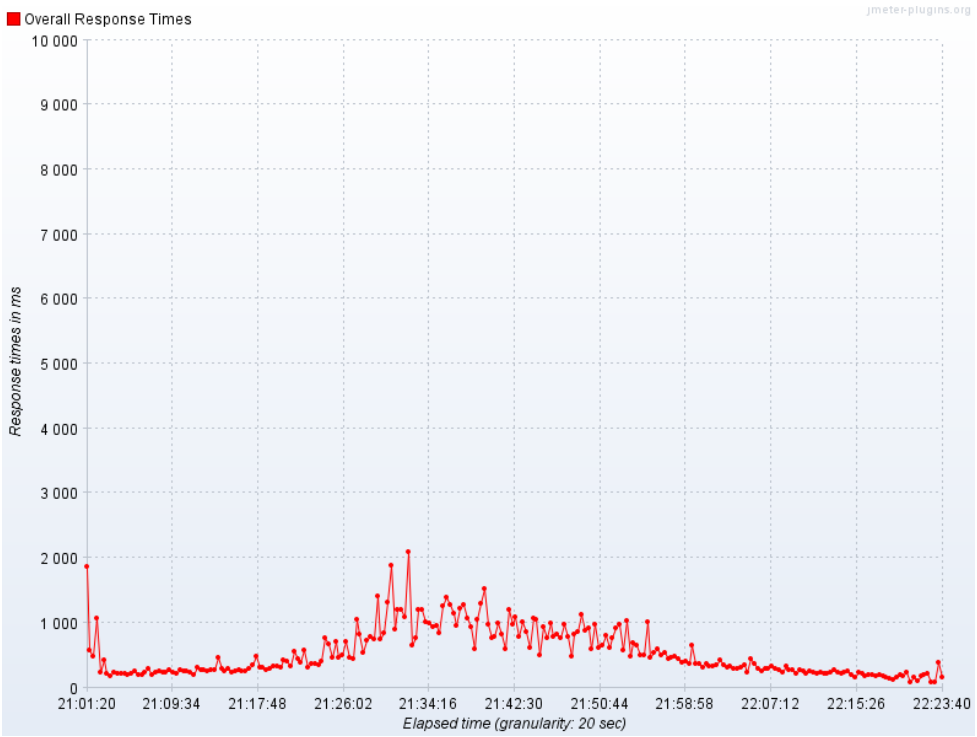


Figure 28 Overall Response Times

In comparing these two graphs it is possible to check multiple results:

- what was the maximum response time in the tested scenario,
- what was the maximum number of the executed requests in one time,
- how long took executing the particular number of responses.

This results can answer also on customer question if the response time does not take longer that was expected. According analysis mentioned in the implementation the time for decision of user is approximately 45 seconds. Test was performed with delay time 7.5 seconds that is six time less. The reason of this reduction is aimed to save testing environment resources. Consideration of using this reduction means six times more users. So if the test processes 300 users (threads), in results it represent 1800 users. This way can save testing cost (additional processor and memory).

7 Discussion

The main objective of this thesis was to design and implement performance test and execute it against an application based on IBM WebSphere Commerce. The test was implemented using performance testing tool Apache JMeter which is a free solution for this approach. The process of creating performance testing can be divided into two parts that are using components provided by testing tools and design specific test scenario. In this thesis were realized both of this part successfully. Designed scenario uses the most common flow where user crawls through web pages, does shopping action and finds product in categories.

This testing was run against application built on IBM WebSphere Commerce system. This is very huge system that is able to process a lot of actions effectively but it has to be well designed. There is a lot of pages caching that enhance performance, shorten response time so doing performance testing can find issues in this sphere too.

Creating a test scenario is not a simply operation. The test should be based on a real user behavior for producing relevant data. Observation of user behaviors and analyzes can improve results from testing. For example that can be decision specified by percentual distribution, most visited parts of web etc. This attempt can be added to future implementation for representing the results more rigorously.

8 References

Cassone, G., Elia, G., Gotta, D., Mola, F., & Pinnola , A. (n.d.). *Web Performance Testing and Measurement.*

Dynamic Testing. Accessed on 2015, April 28. Retrieved from Dynamic Testing: http://www.tutorialspoint.com/software_testing_dictionary/dynamic_testing.htm

Erinle, B. (2013). *Performance Testing with JMeter 2.9.* Birmingham: Packt Publishing Ltd.

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2007). *Software Testing.* Swindon: BCS.

Integration testing. Accessed on 2015, April 28. Retrieved from Integration testing: http://softwaretestingfundamentals.com/integration-testing/

Integration testing. Accessed on 2015, April 28. Retrieved from Integration testing: http://softwaretestingfundamentals.com/integration-testing/

Java EE & Java Web Learning. Accessed on 2015, April 28. Retrieved from Java EE & Java Web Learning: https://netbeans.org

Meier, J., Farre, C., Bansode, P., Barber, S., & Rea, D. (2007). *Performance Testing Guidance for Web Applications.* Microsoft Corporation.

Software testing. Accessed on 2015, April 28. Retrieved from Software testing: http://en.wikipedia.org/wiki/Software_testing

System testing. Accessed on 2015, April 28. Retrieved from System testing: http://softwaretestingfundamentals.com/system-testing/

Unit testing. Accessed on 2015, April 28. Retrieved from Unit testing: http://en.wikipedia.org/wiki/Unit_testing

Unit testing. Accessed on 2015, April 28. Retrieved from Unit testing: http://softwaretestingfundamentals.com/unit-testing/

WebSphere Commerce application layers. Accessed on 2015, April 28. Retrieved from WebSphere Commerce application layers: http://www-01.ibm.com/support/knowledgecenter/SSZLC2_7.0.0/com.ibm.commerce.developer.doc/concepts/csdapplication.htm

WebSphere Commerce common architecture. Accessed on 2015, April 28. Retrieved from WebSphere Commerce common architecture: http://www-01.ibm.com/support/knowledgecenter/SSZLC2_7.0.0/com.ibm.commerce.developer.doc/concepts/csdsoftwarecomp.htm

WebSphere Commerce product overview. Accessed on 2015, April 28. Retrieved from WebSphere Commerce product overview: http://www-01.ibm.com/support/knowledgecenter/SSZLC2_7.0.0/com.ibm.commerce.admin.doc/concepts/covoverall.htm

White box testing. Accessed on 2015, April 28. Retrieved from White box testing: http://softwaretestingfundamentals.com/white-box-testing/