**CMPE273: Enterprise Distributed Systems**

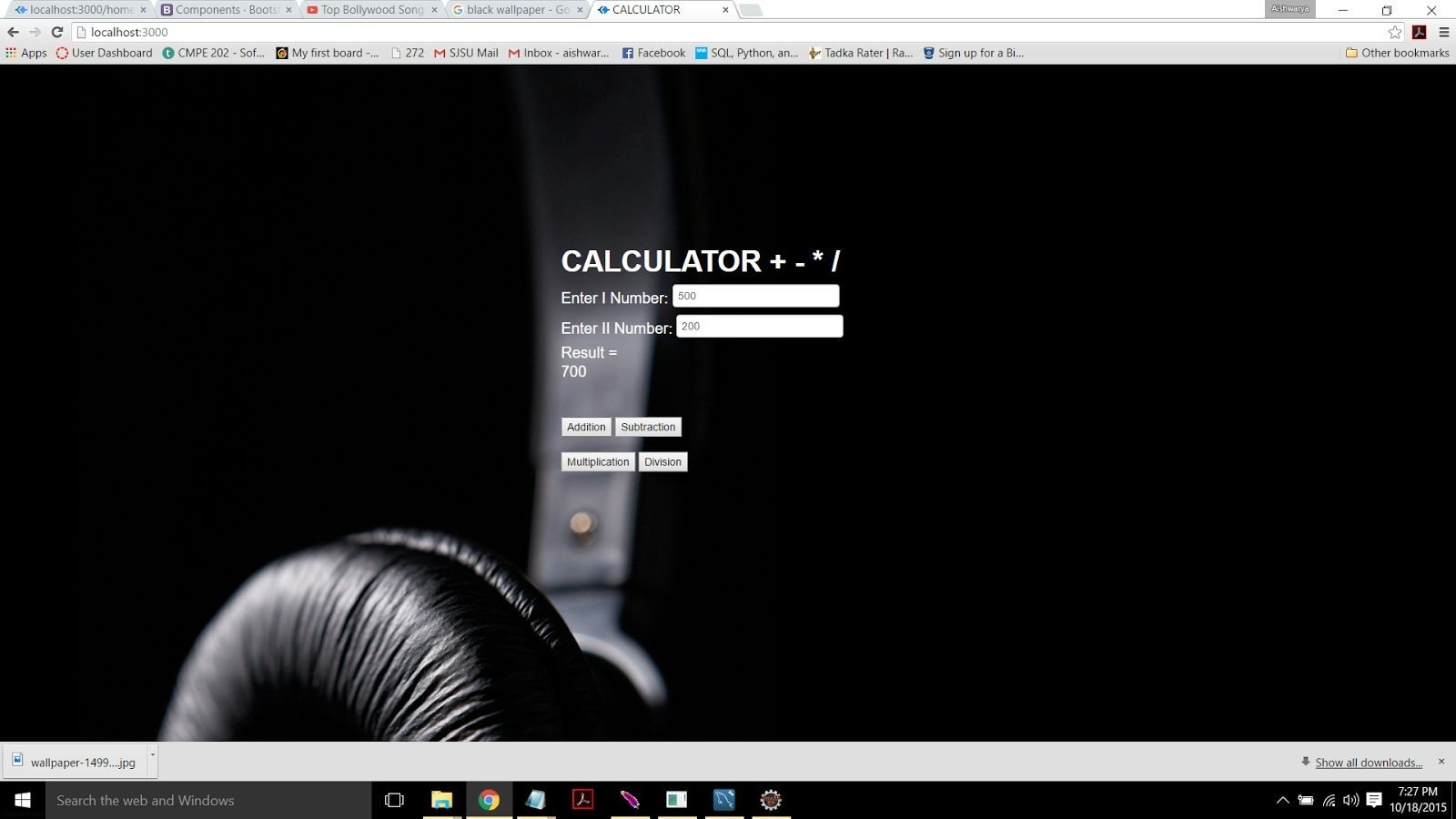
**Lab 1 Assignment: Using REST (Node.js) and Angular JS Name : Patwardhan Aishwarya**

**Id : 010735673**

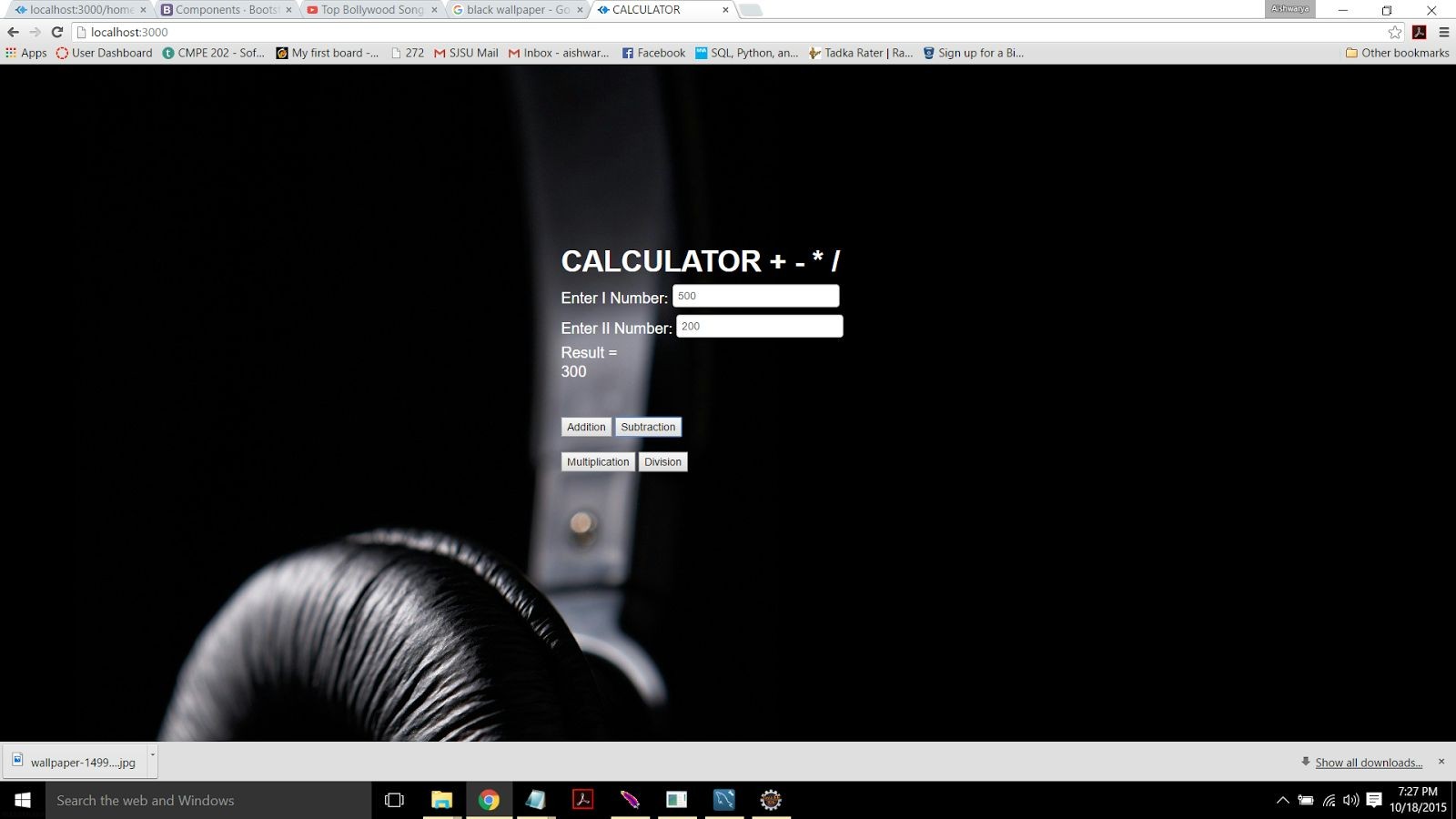
Server 1 ­ “Calculator” to demonstrate Stateless web services

The first Node.js based server you need to develop is the “Calculator”. This server should perform the following tasks:

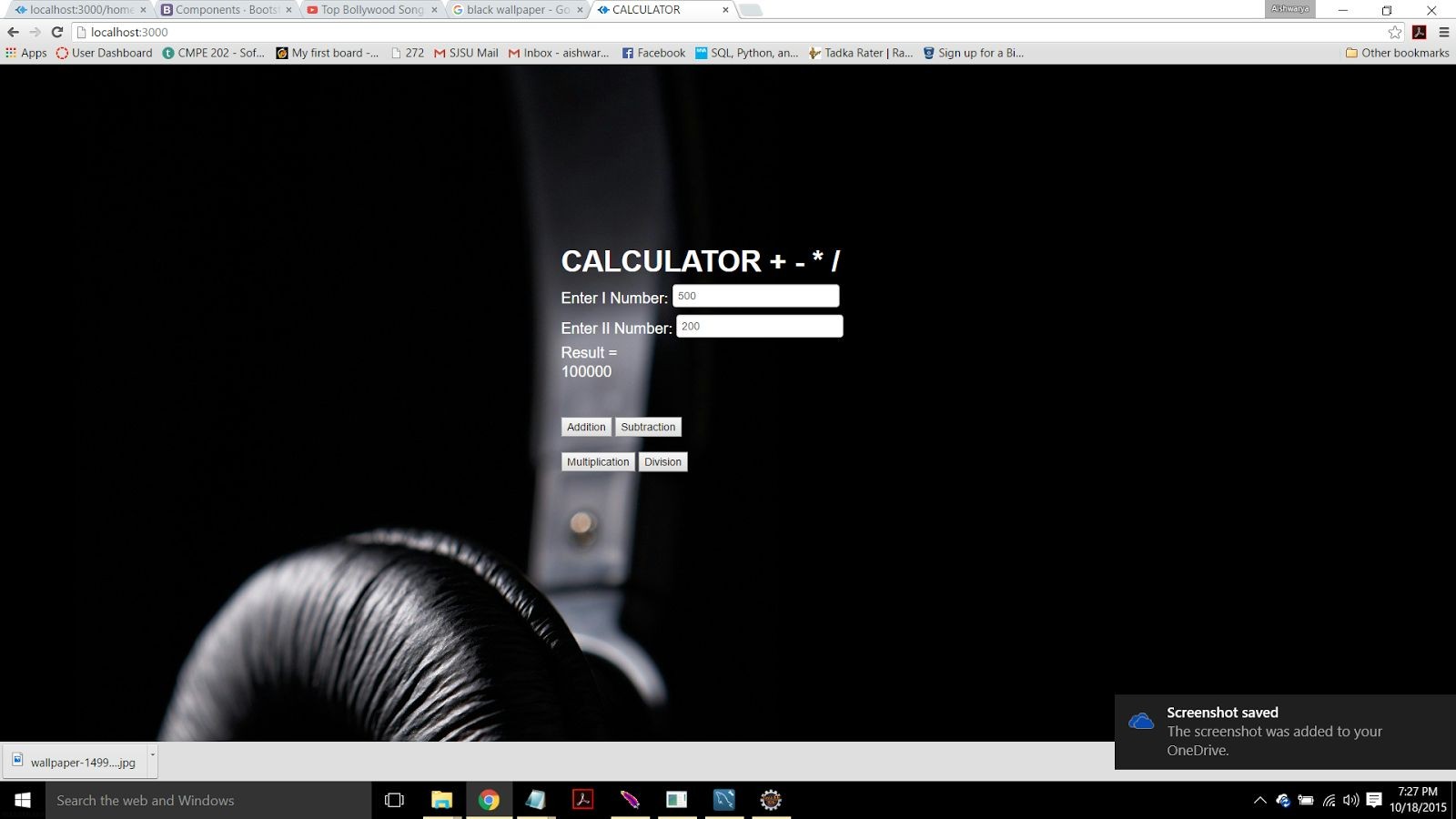
1. Addition



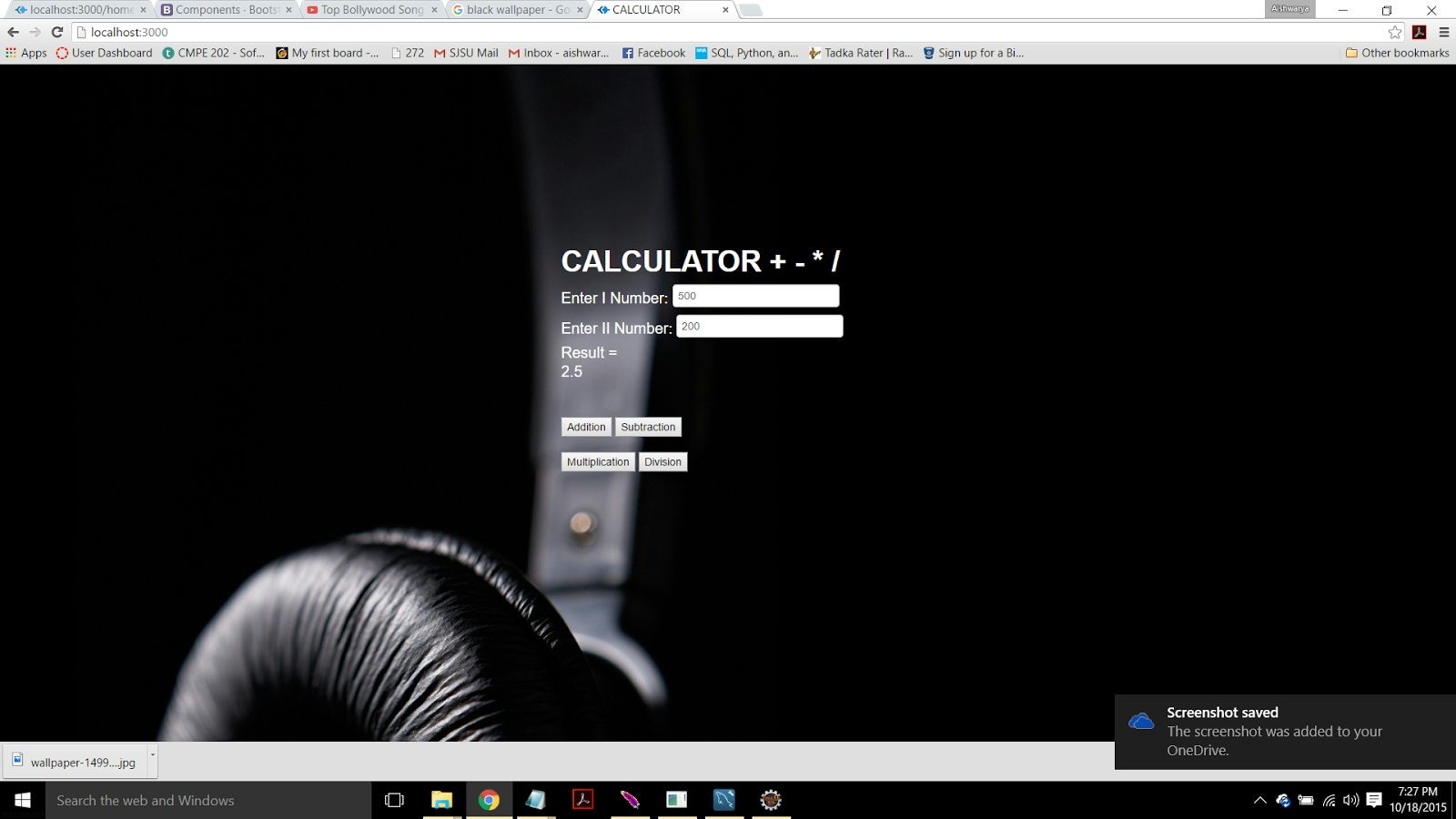
2. Subtraction



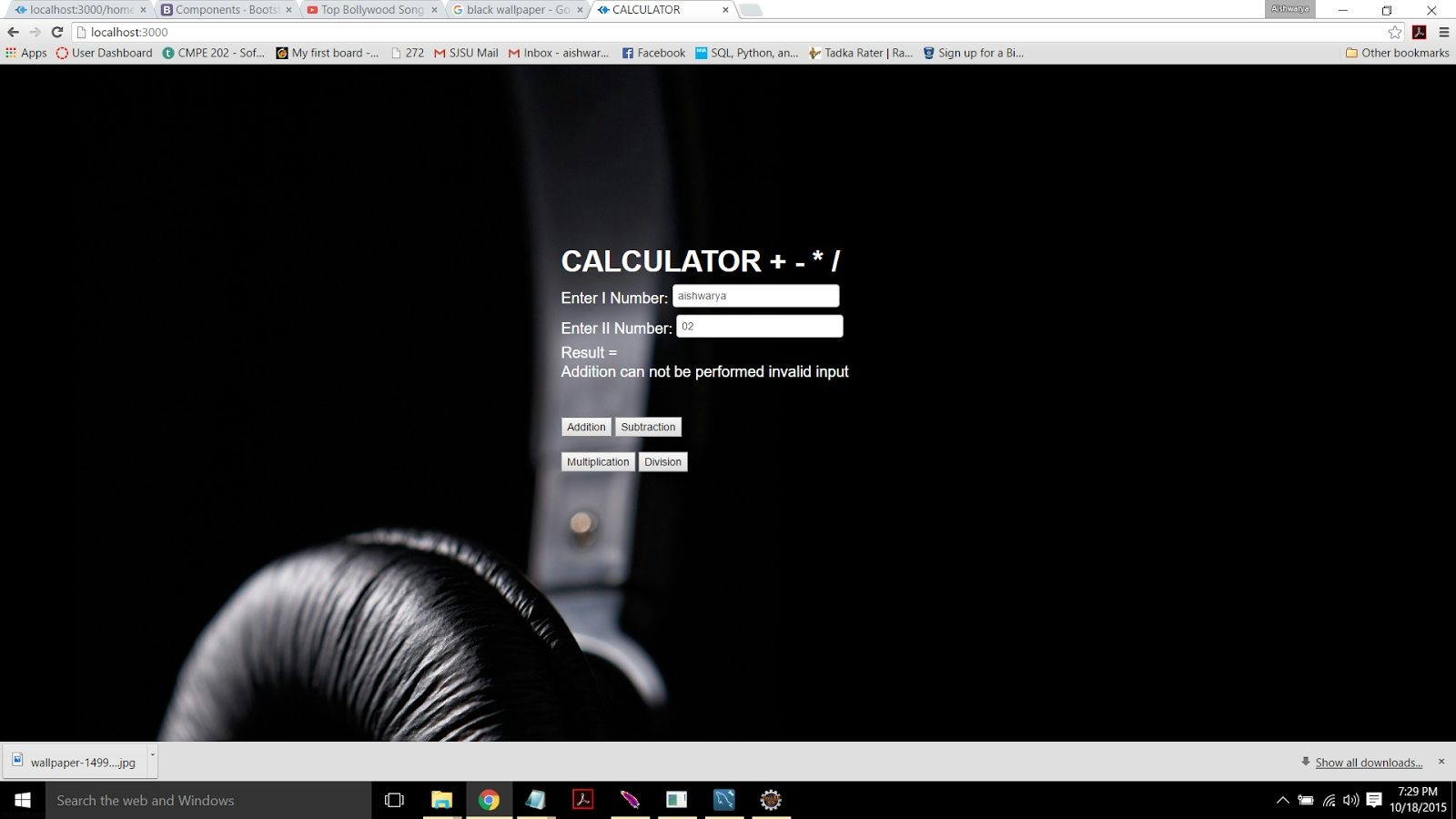
3. Multiplication



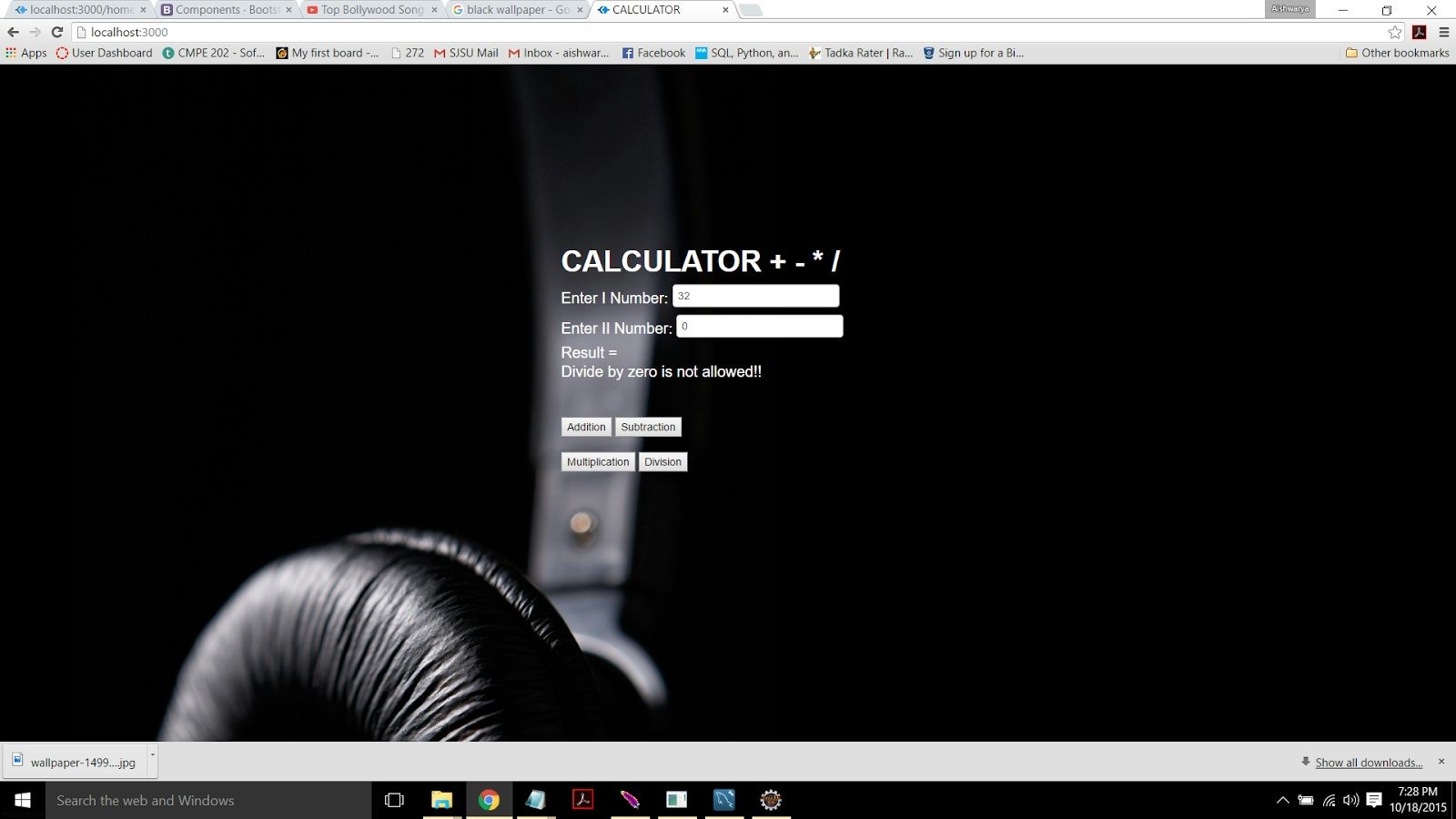
4. Division



Take care of exceptions. EXCEPTIONS : STRING INPUT:



DIVIDE BY ZERO EXCEPTION:



Client 1 – “ Calculator client ” Client can interact with java script.

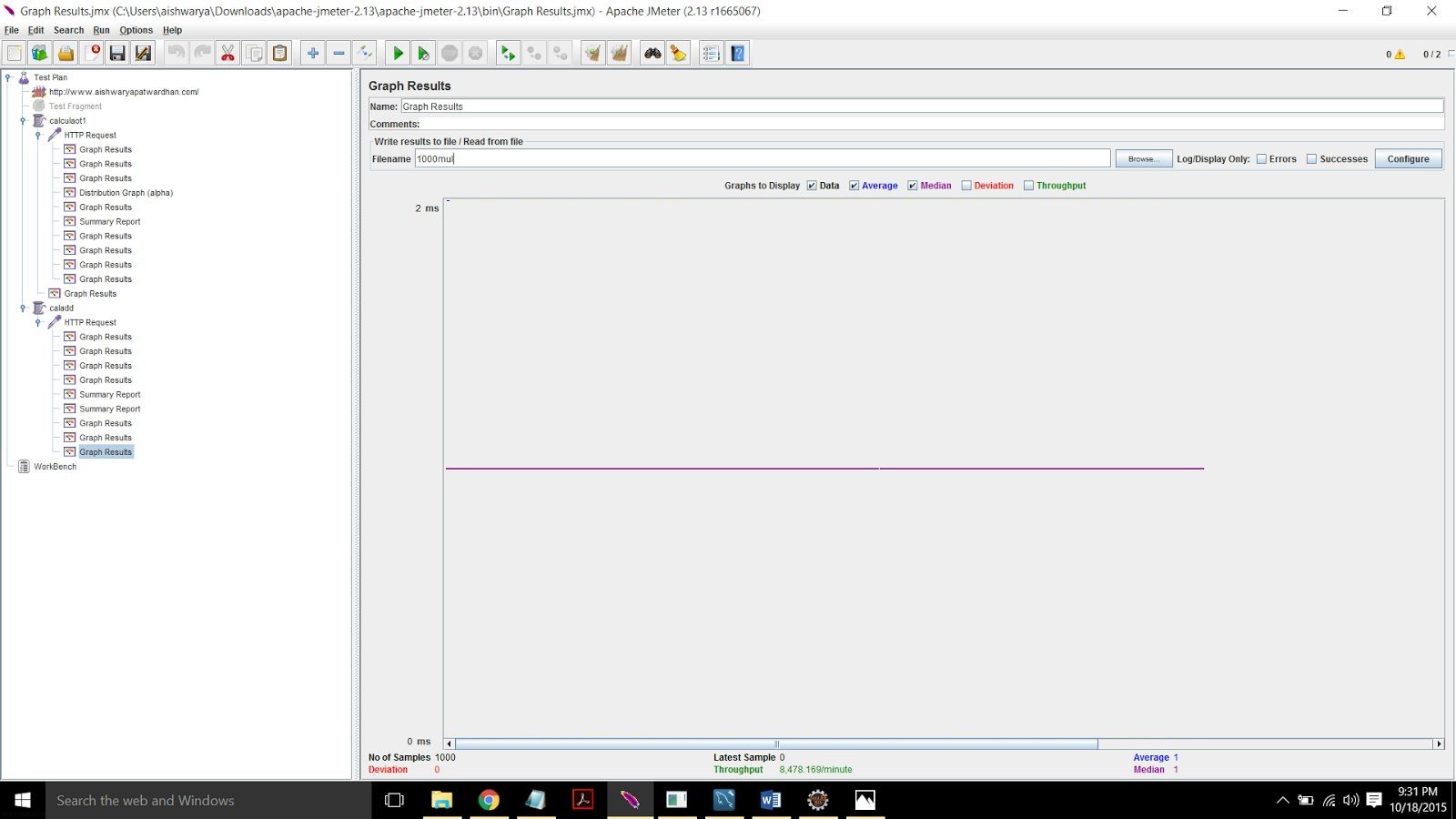
The client should provide the facility to test all the functions of calculator service. The client for Server 1 should behave in the following way:

Perform each operation of Server once. Print out the result in a systematic format, then automate the following process and print the average times as mentioned below.

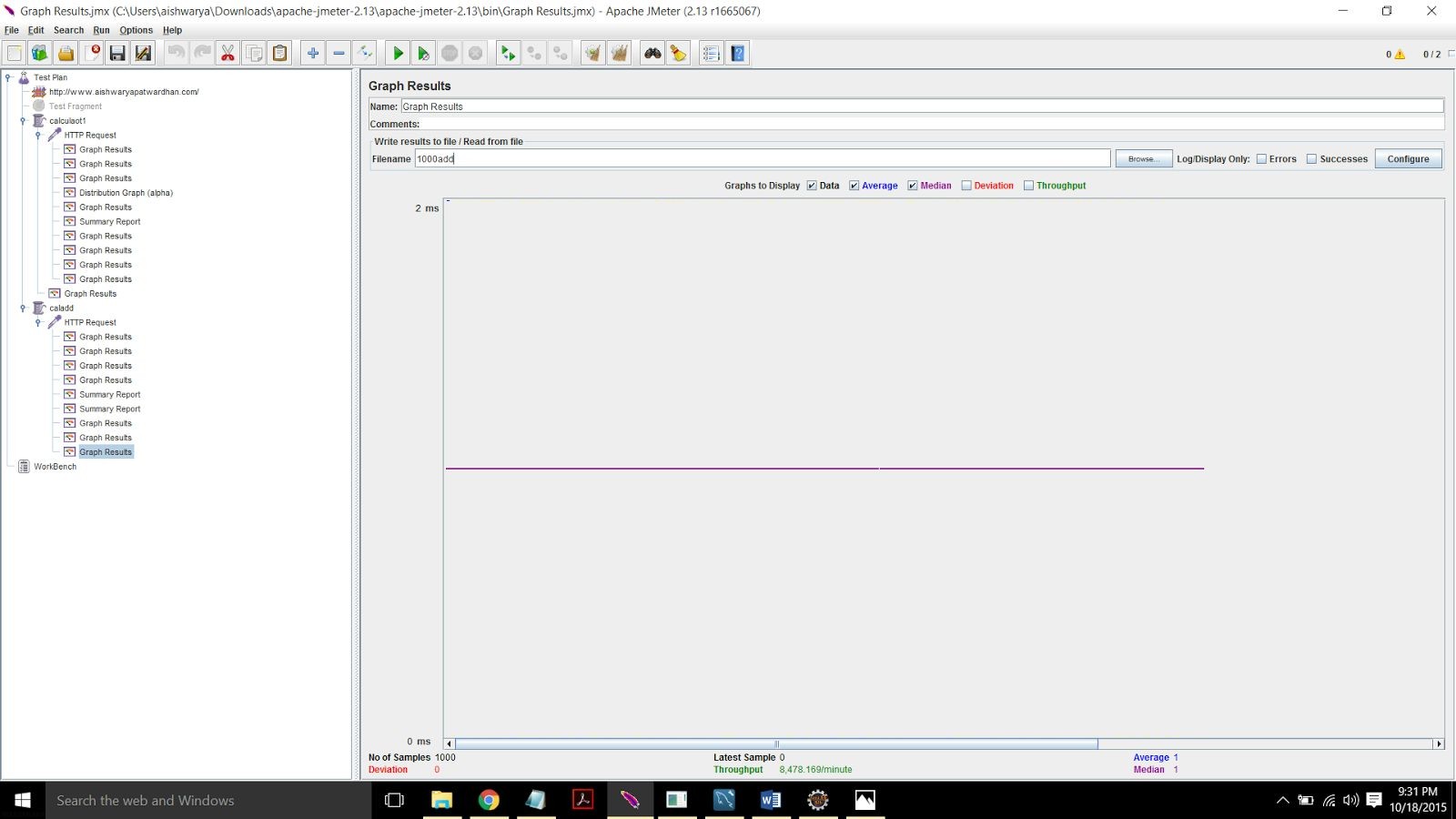
1. Start a timer

2. Invoke 1,000 calculator calls on randomly selected tasks.

3. Stop the timer and print out the average time to perform each operation



1000 loop count for multiplication function.

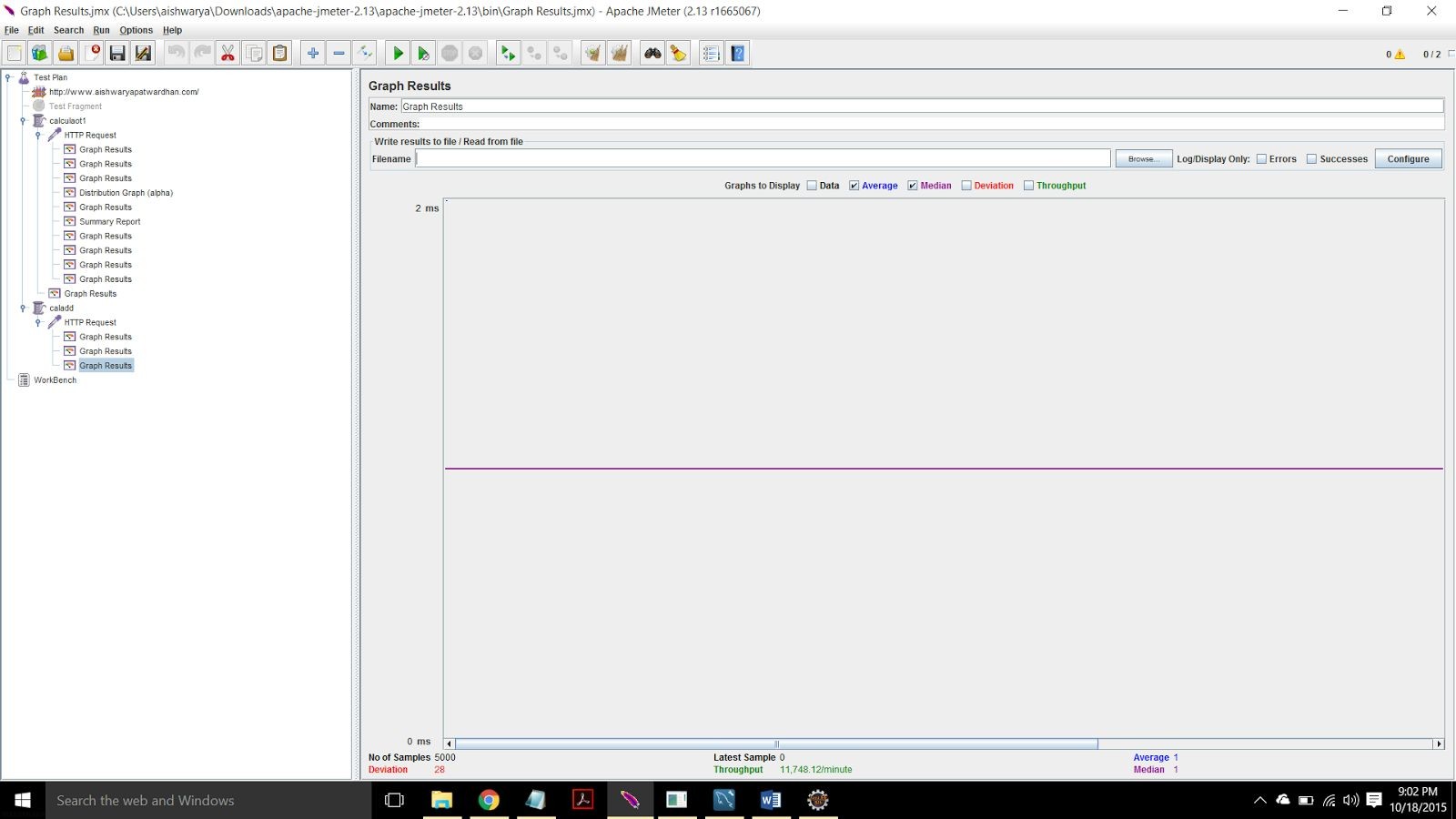


1000 loop count for addition function.

1. Start a timer

2. Invoke 5,000 calculator calls on randomly selected tasks.

3. Stop the timer and print out the average time to perform each operation

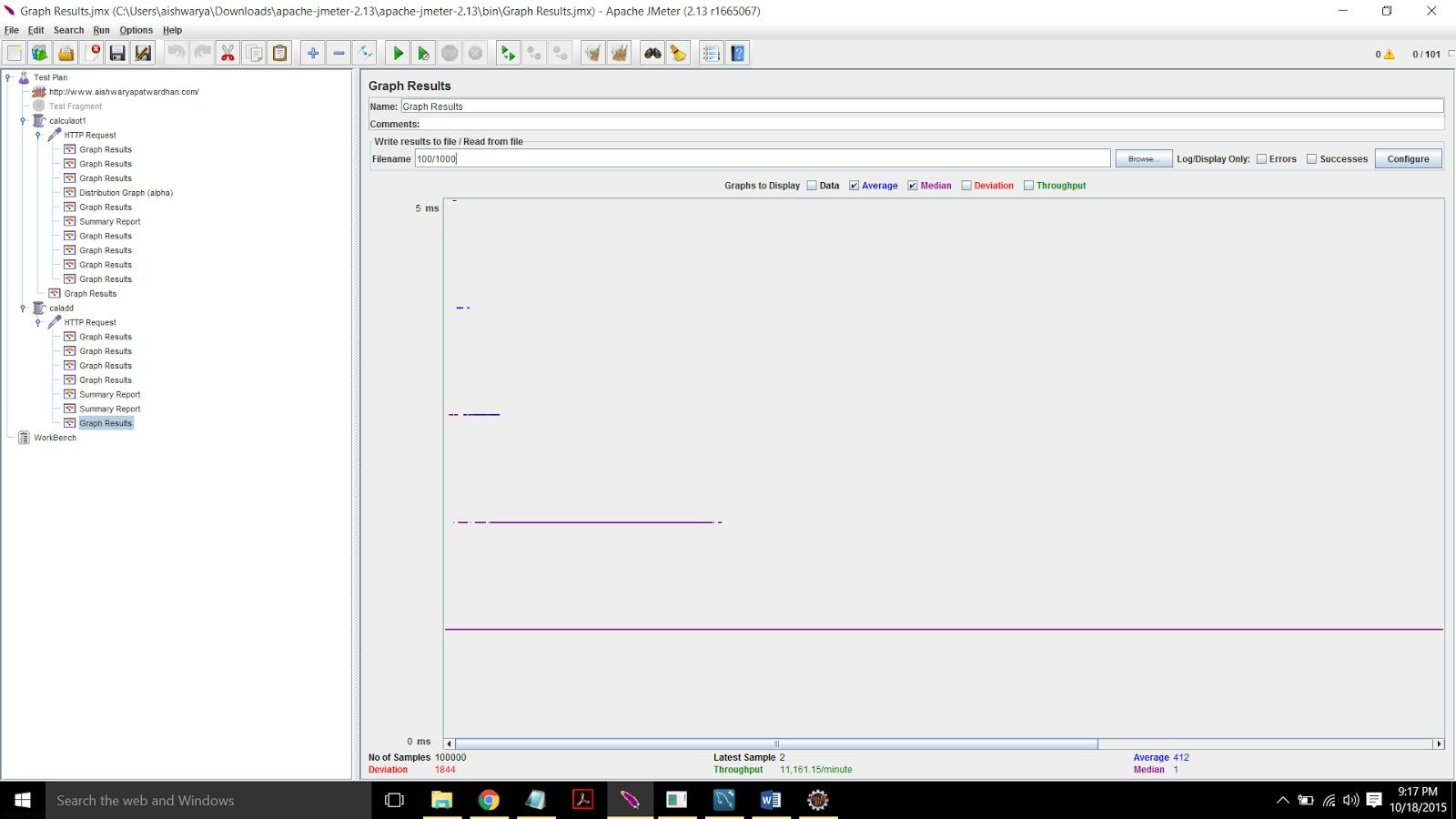


5000 loop count for multiplication function.

1. Start a timer

2. Invoke 100 concurrent users with 1000 calls each to calculator on randomly selected tasks.

3. Stop the timer and print out the average time to perform each operation



Comparison :

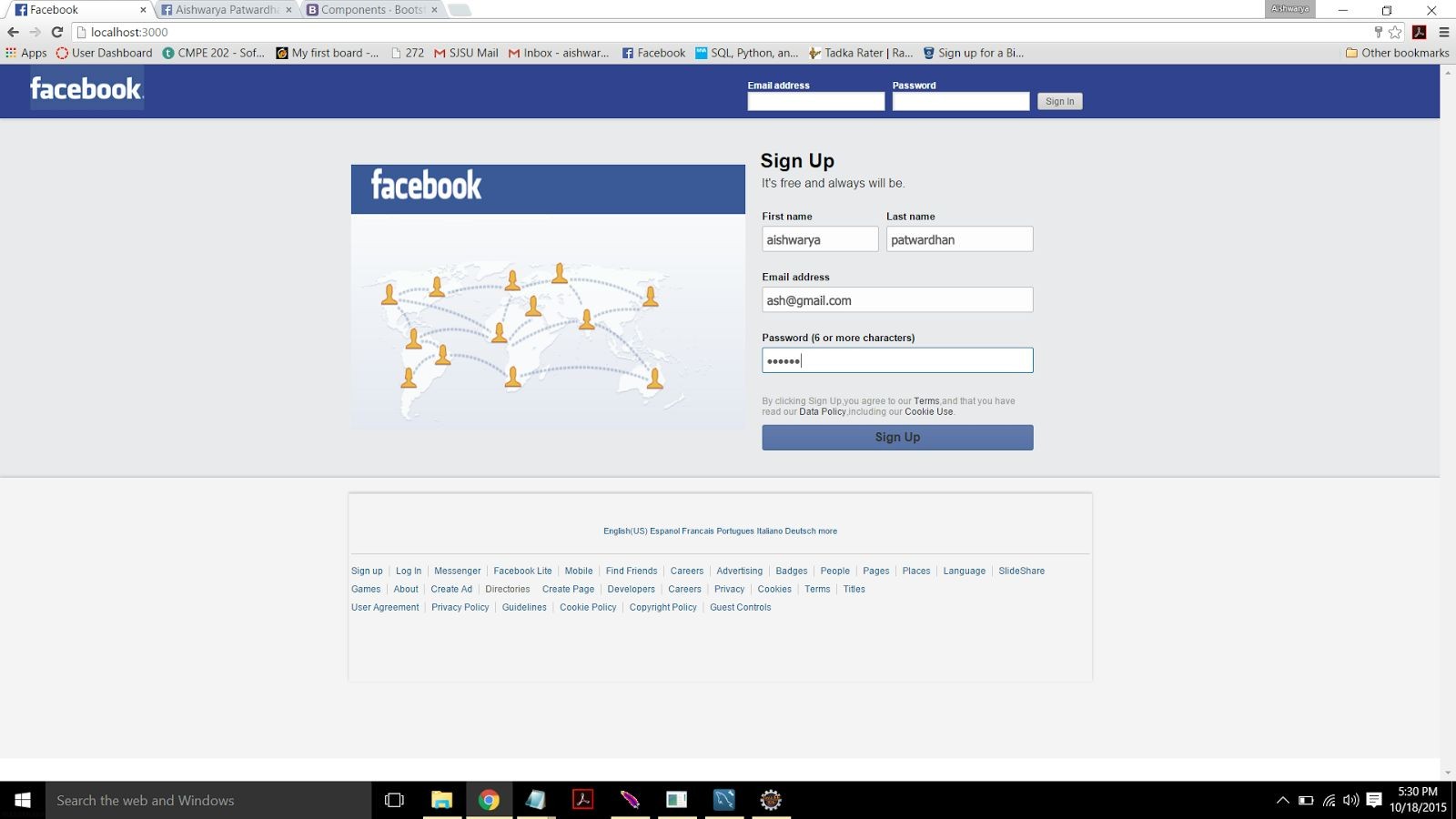
As you can see from the graph that when loop count is 1000 and 5000 there is not much change in throughput and average but when the number of users are increased from 1 to 100 and done with 1000 loop count throughput count increases drastically.

­­\*\*­­

Server 2 ­ “Facebook application” to demonstrate RESTful Services

The next node.js based server you need to develop is the “Prototype of Facebook application”.

This server s



hould perform the following tasks:

a) Basic Users functionalities:

1. Sign up new user (at least first name, last name, Email and password)

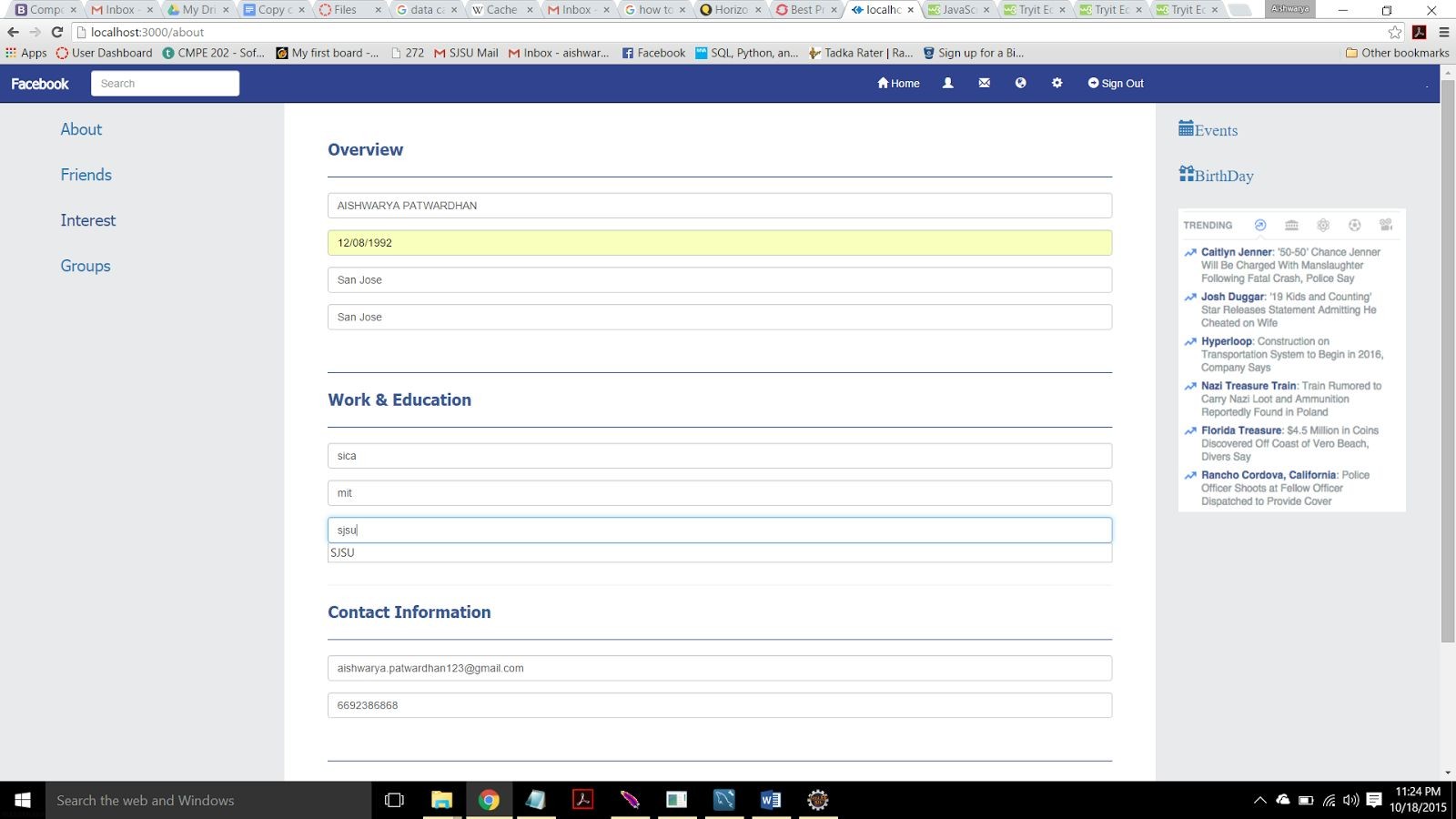
2. Sign in existing user

3. Sign out.

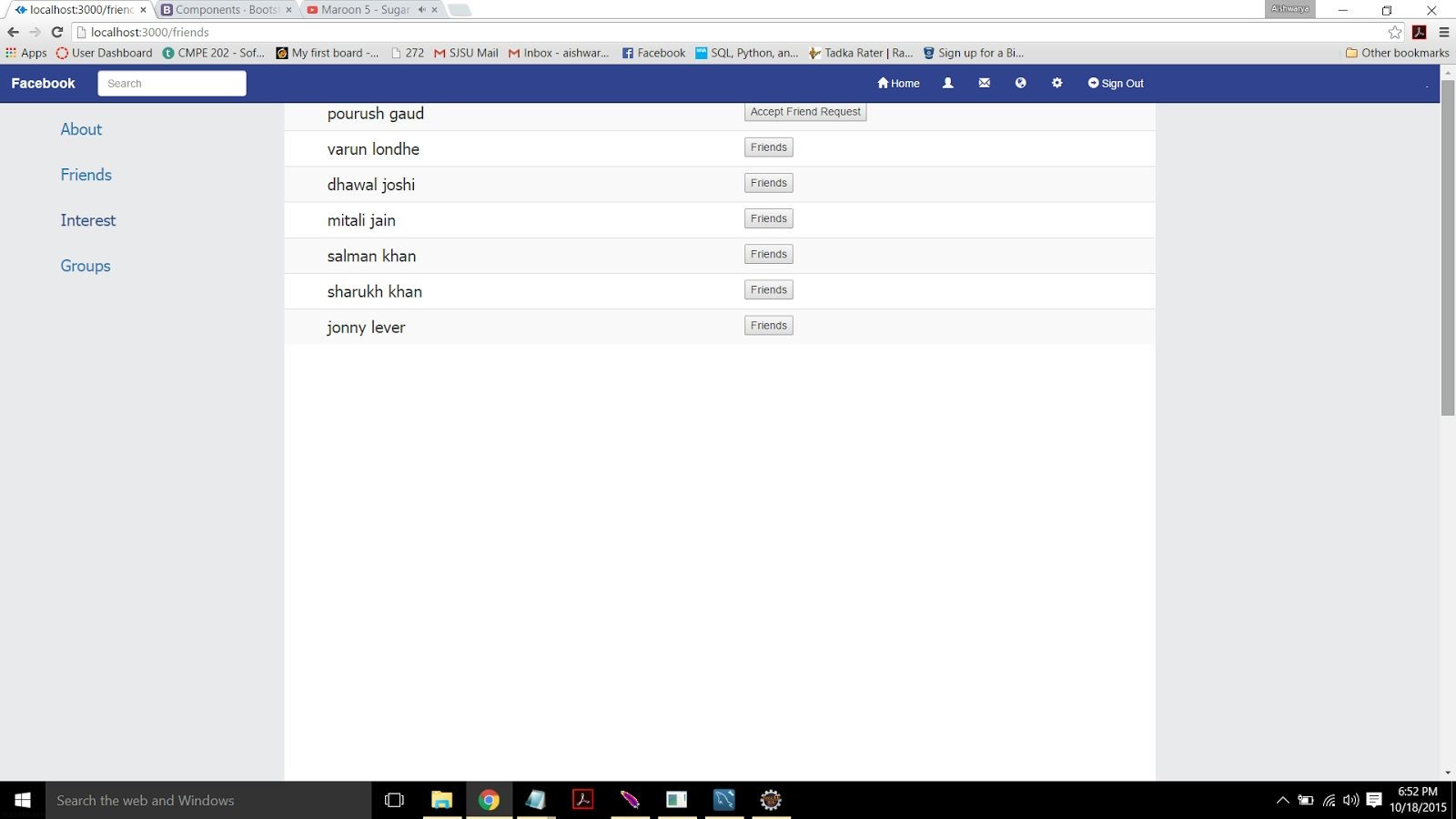
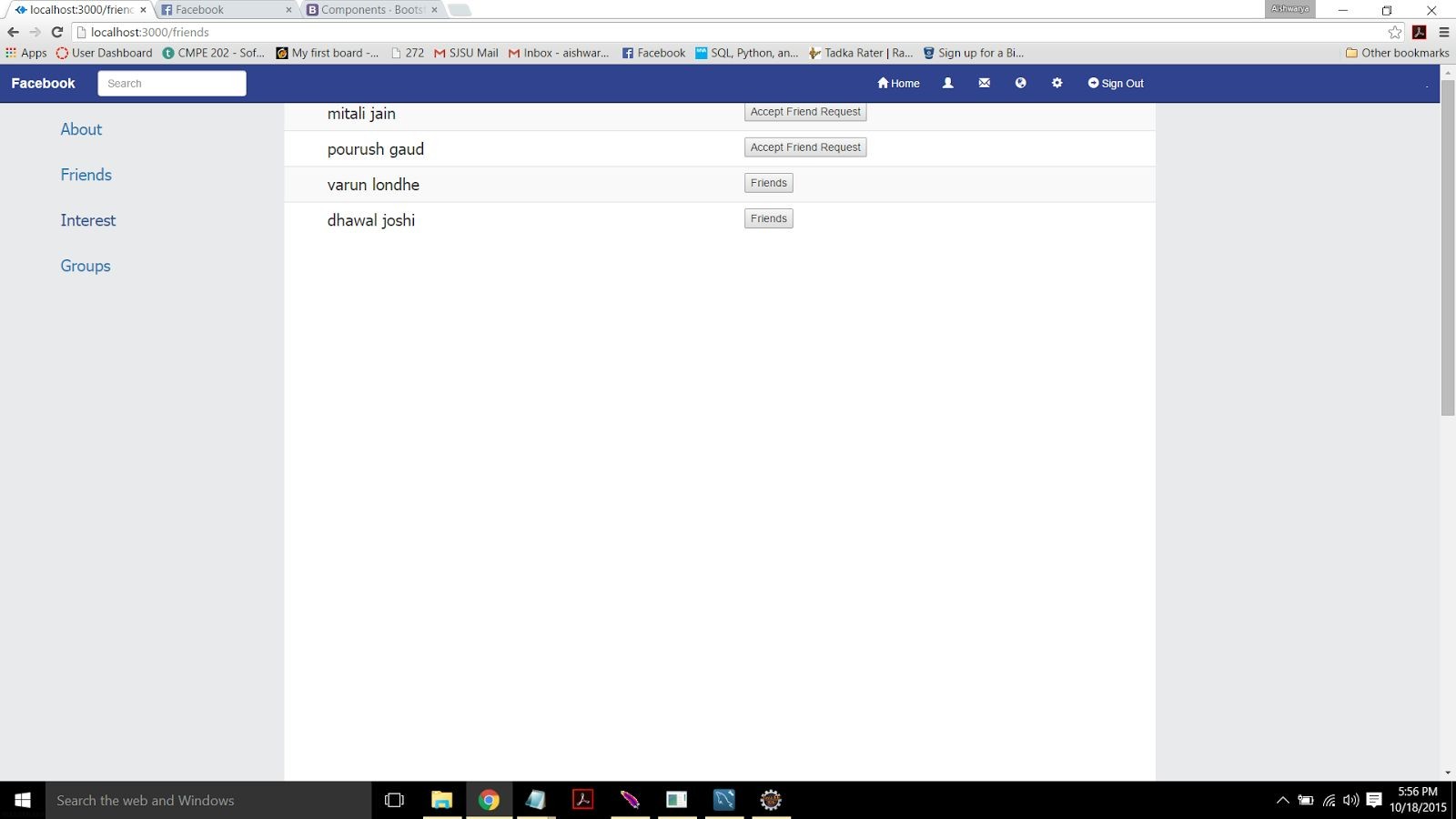
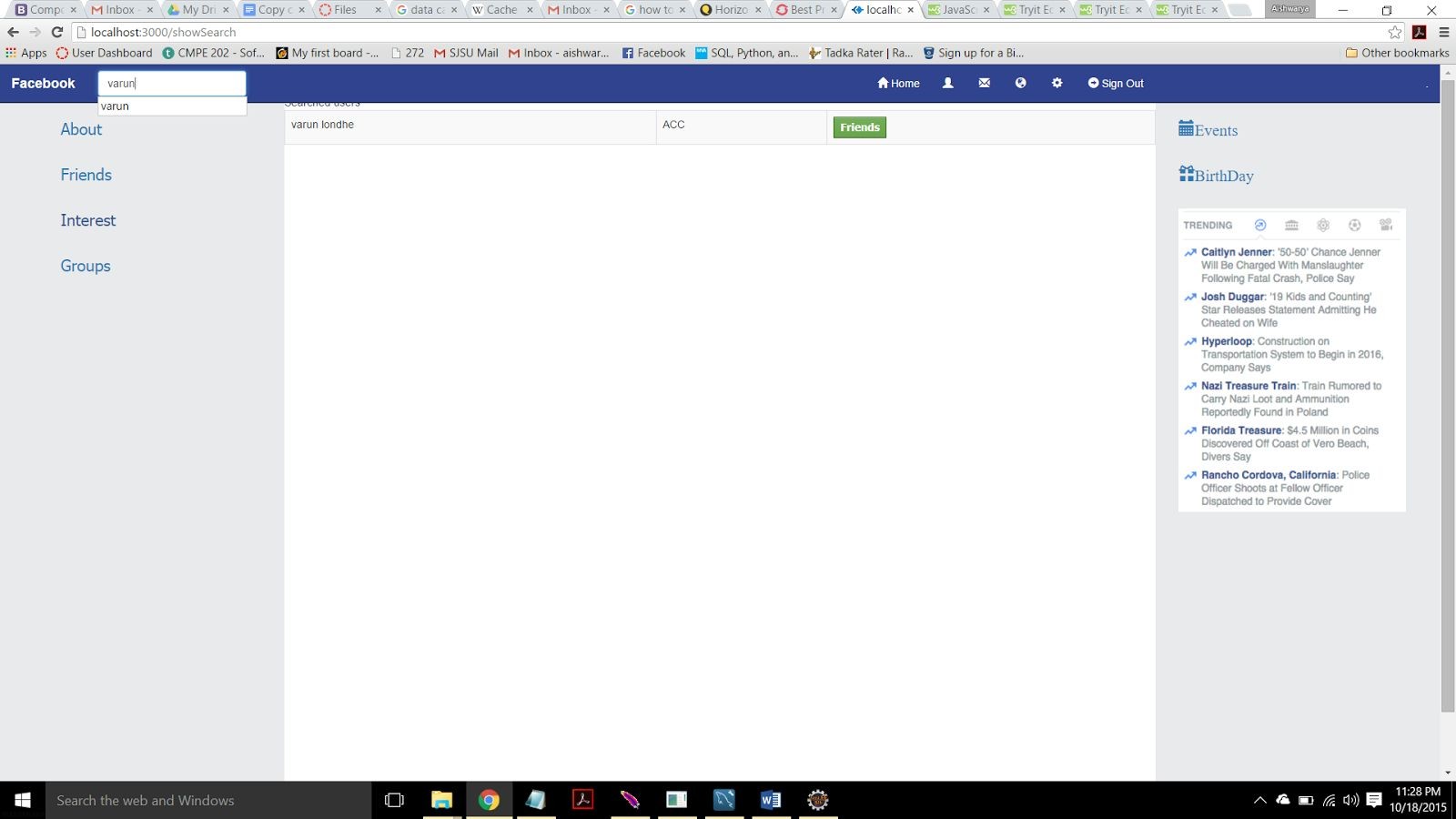
b) Users account should provide basic details such as:

1. About:

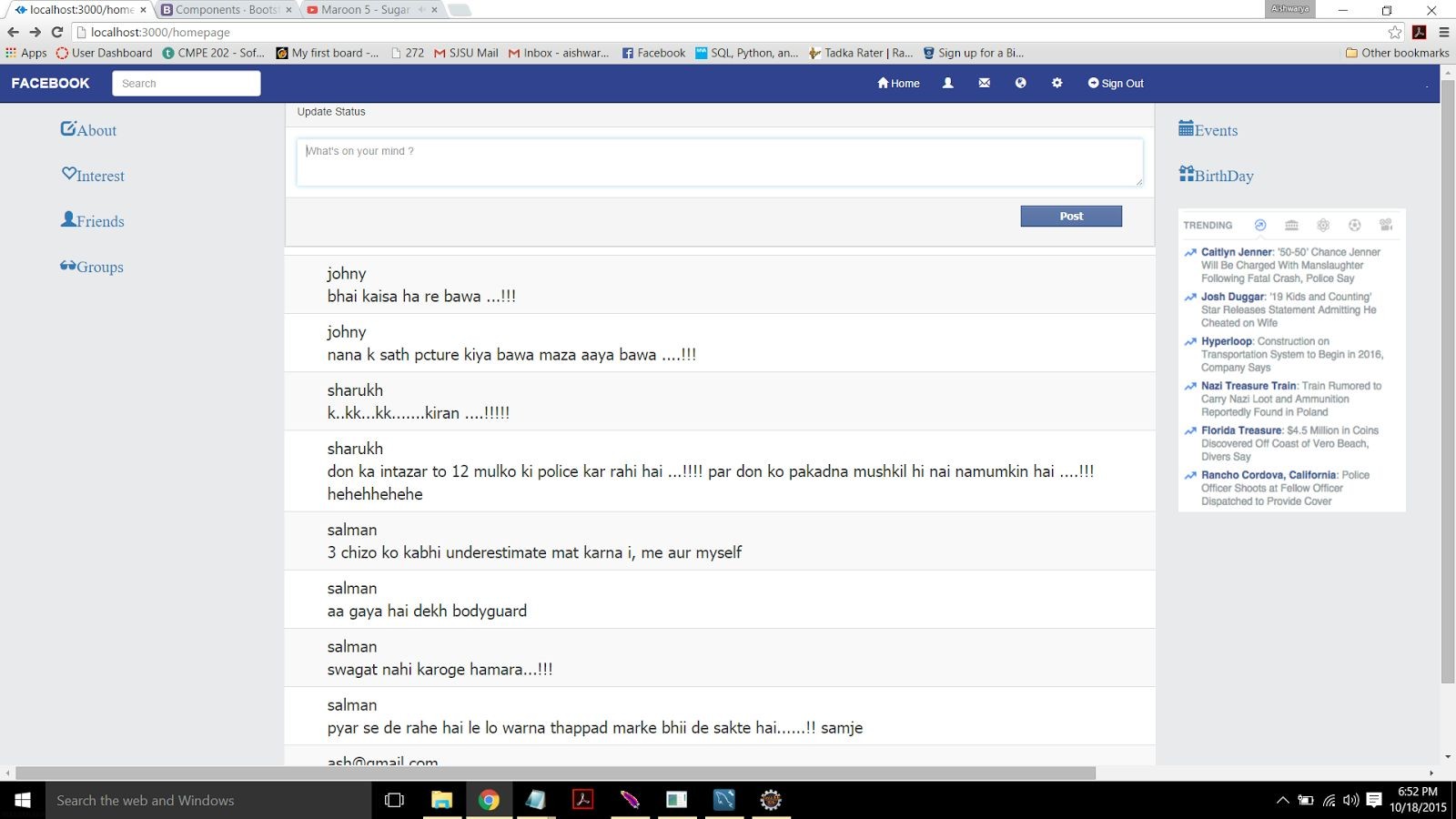
User overview, Work and education, contact info and life events.



2. Friends list: Send friend request, accept friend request. search friend

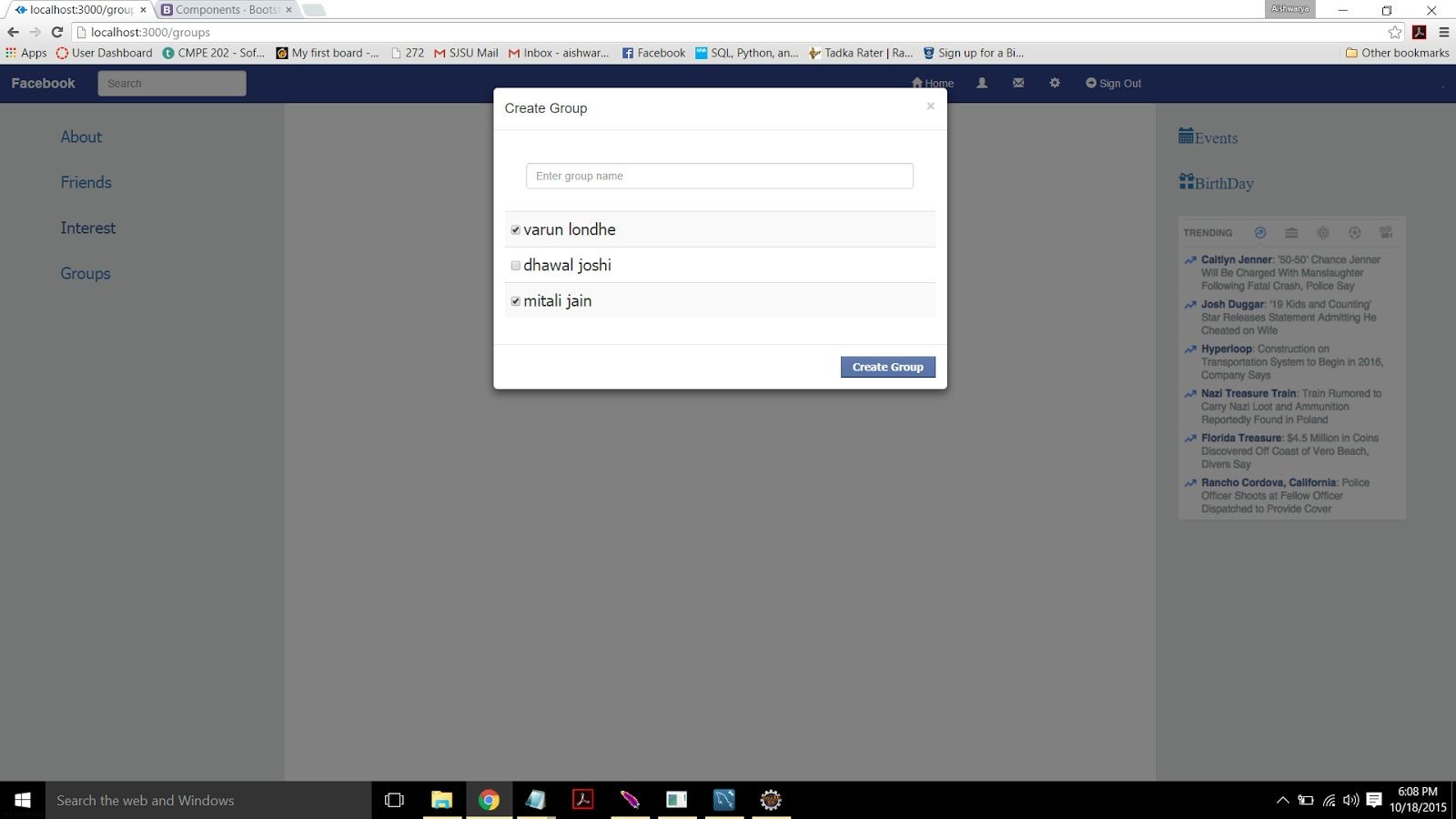


c) Provide news feed functionality.

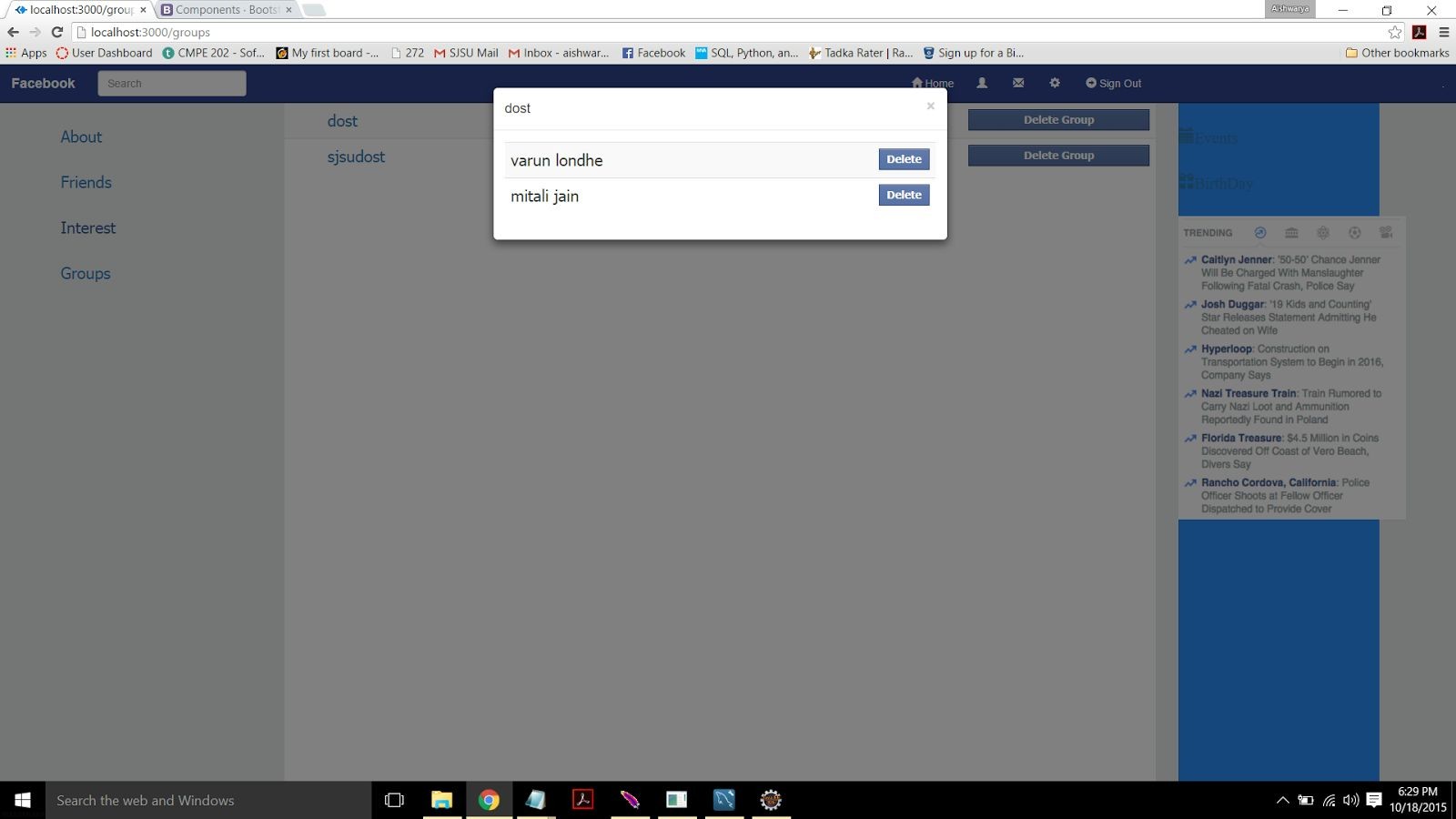


d) Provide Groups functionalities:

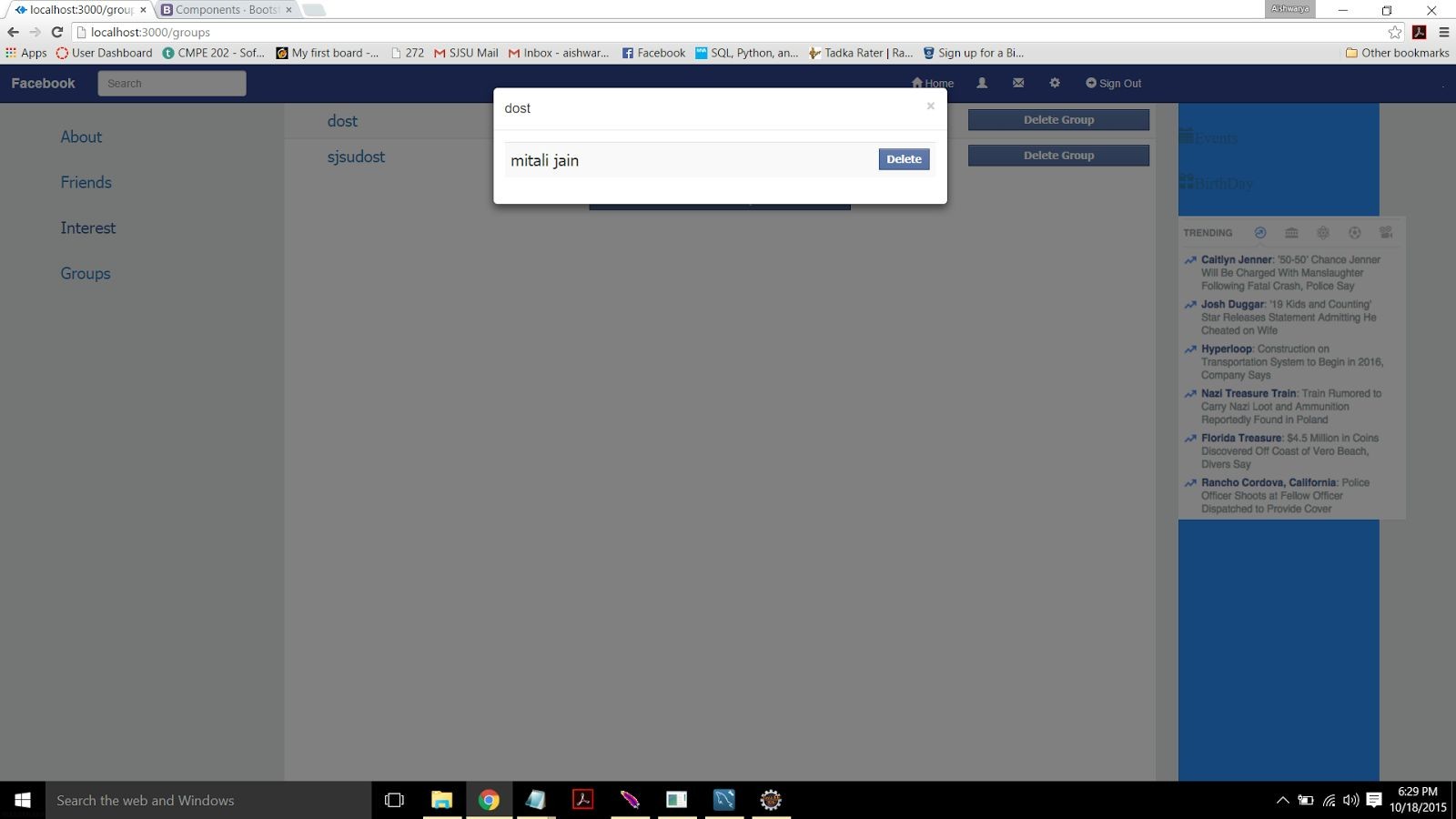
1. Create group 2. Add member in a group



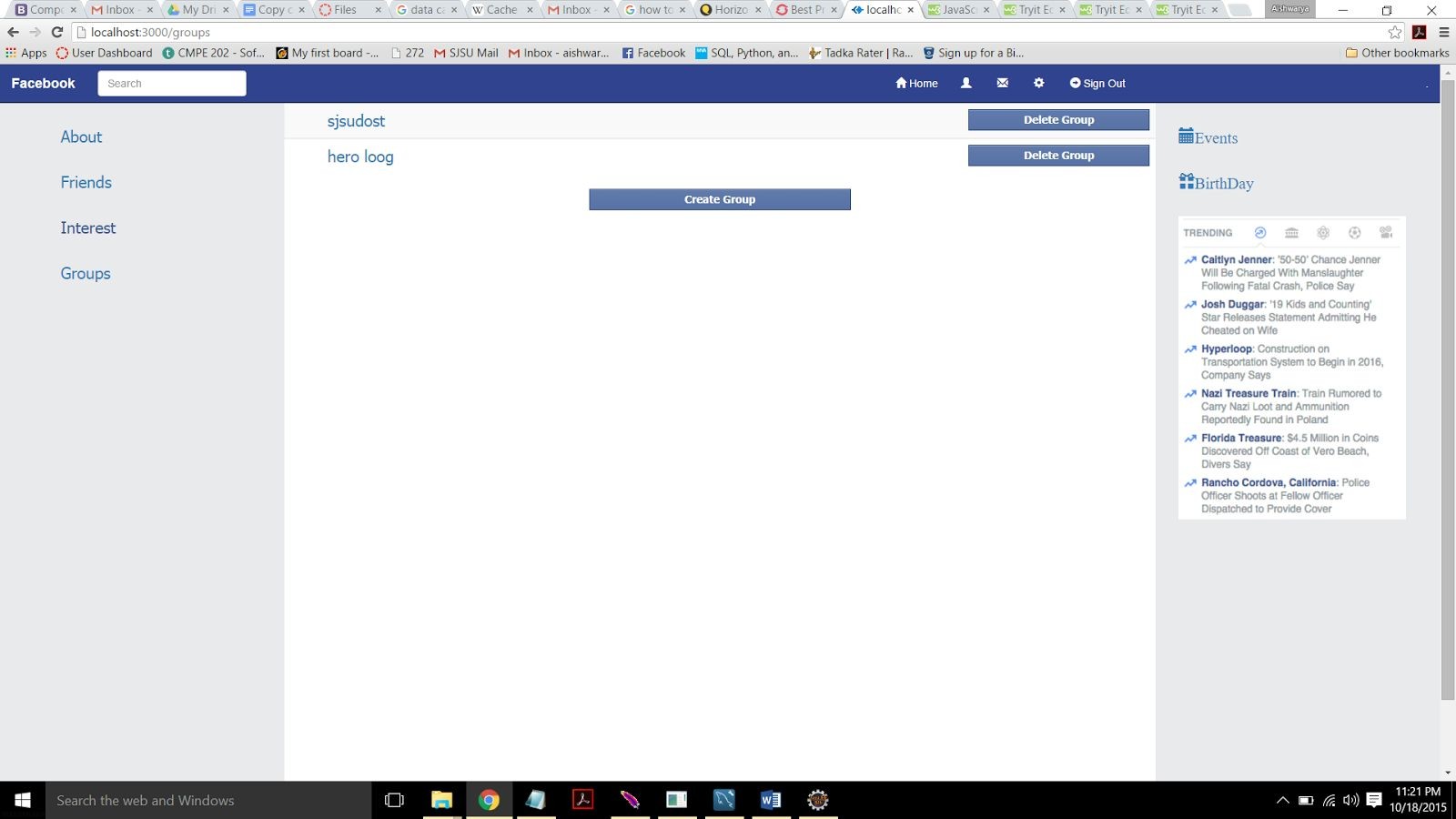
3. Show members in group



4. Delete member from a group



5. Delete group



e) Should perform connection pooling for database access.

The Service should take care of exception that means validation is extremely important for this server. Good exception handling and prototype similar to actual Facebook application would attract good marks.

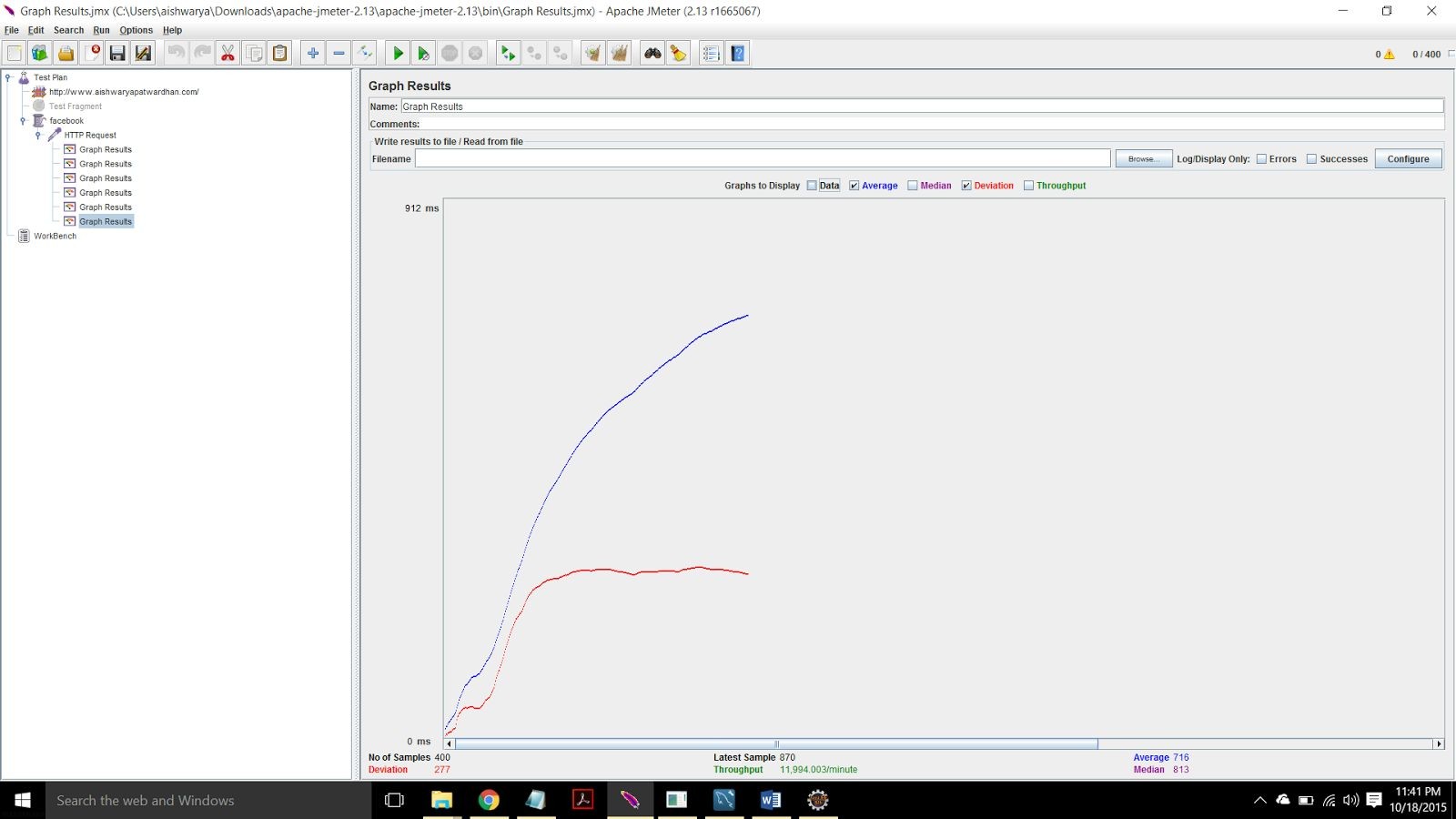
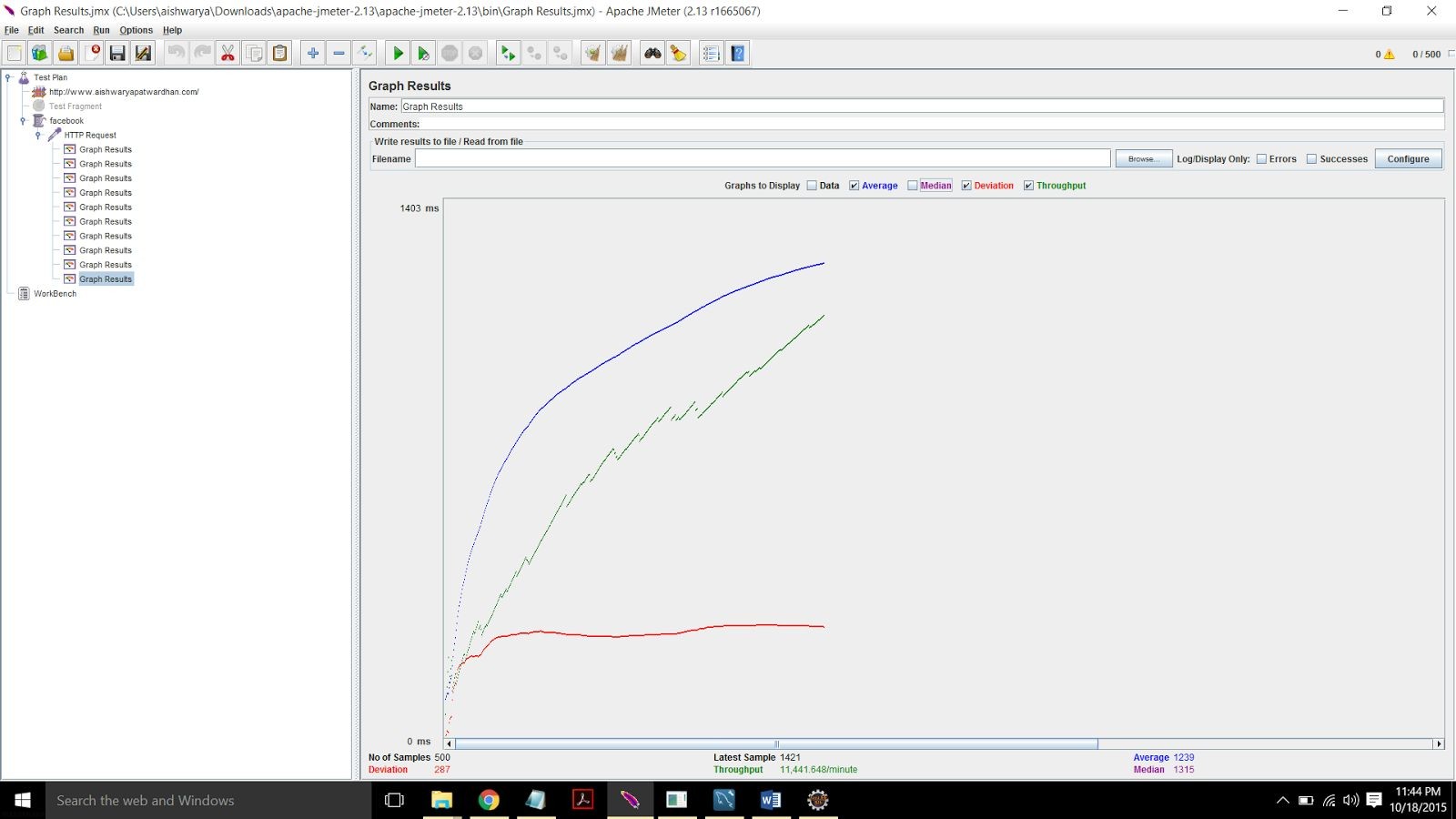
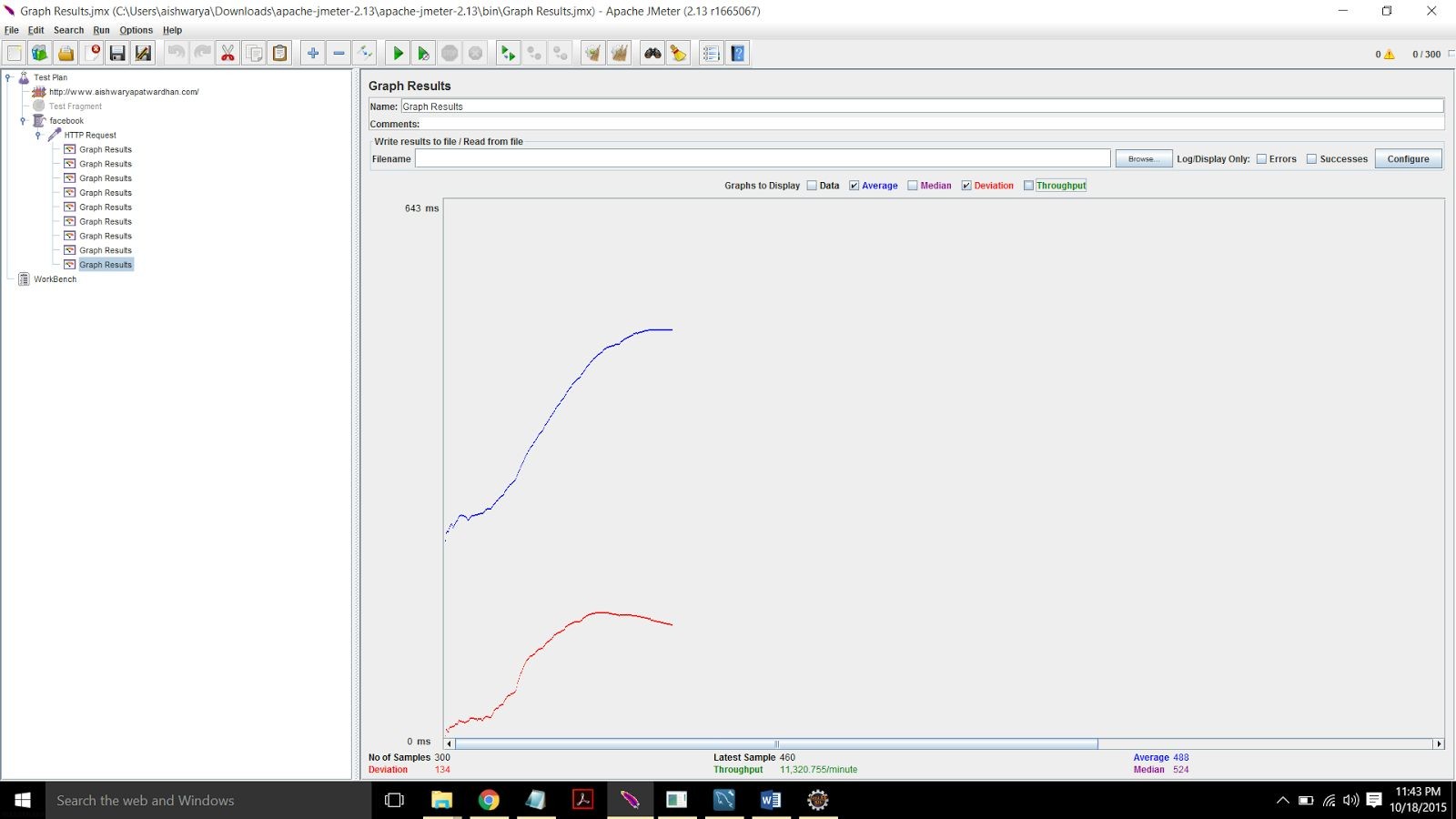
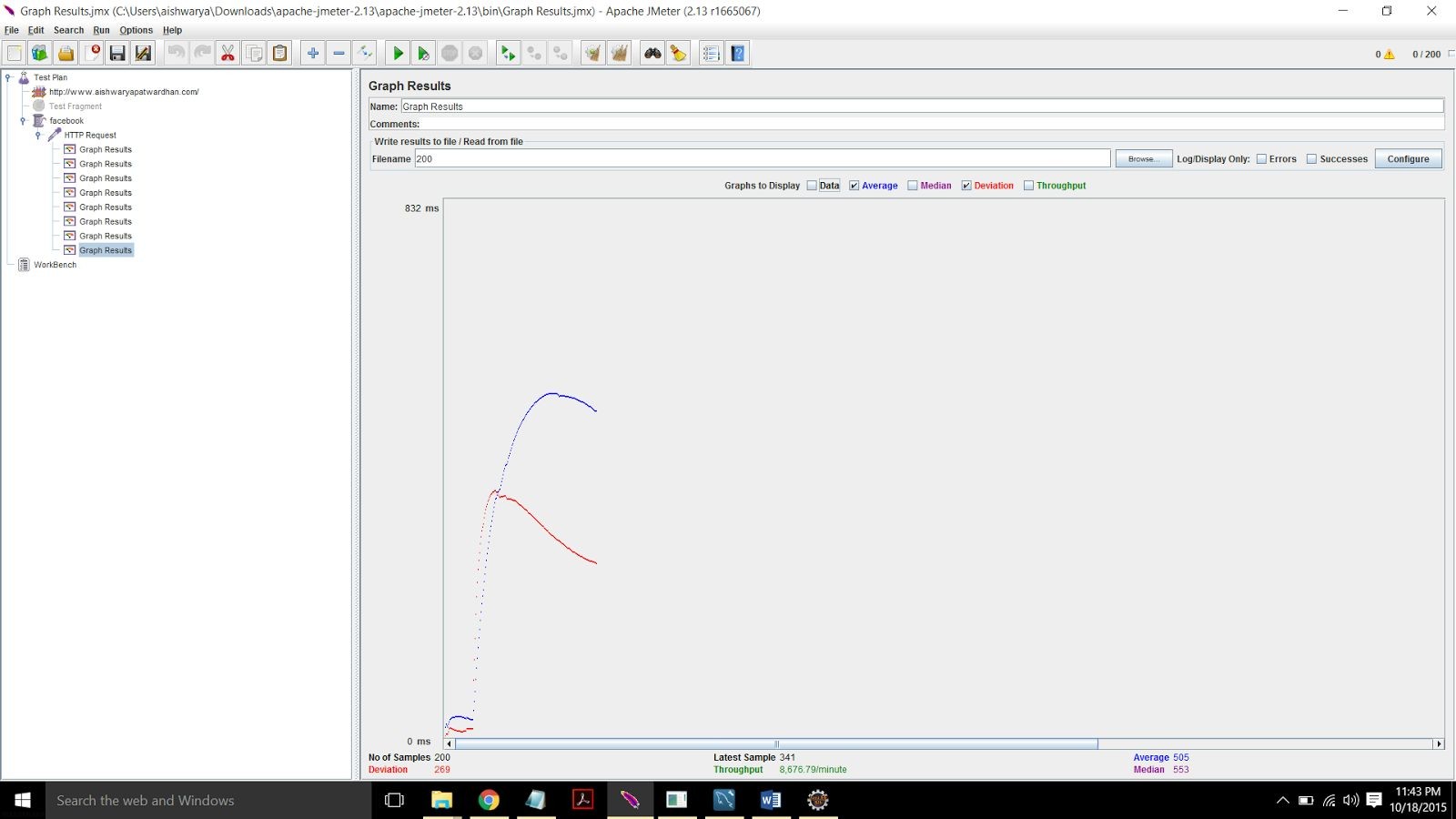
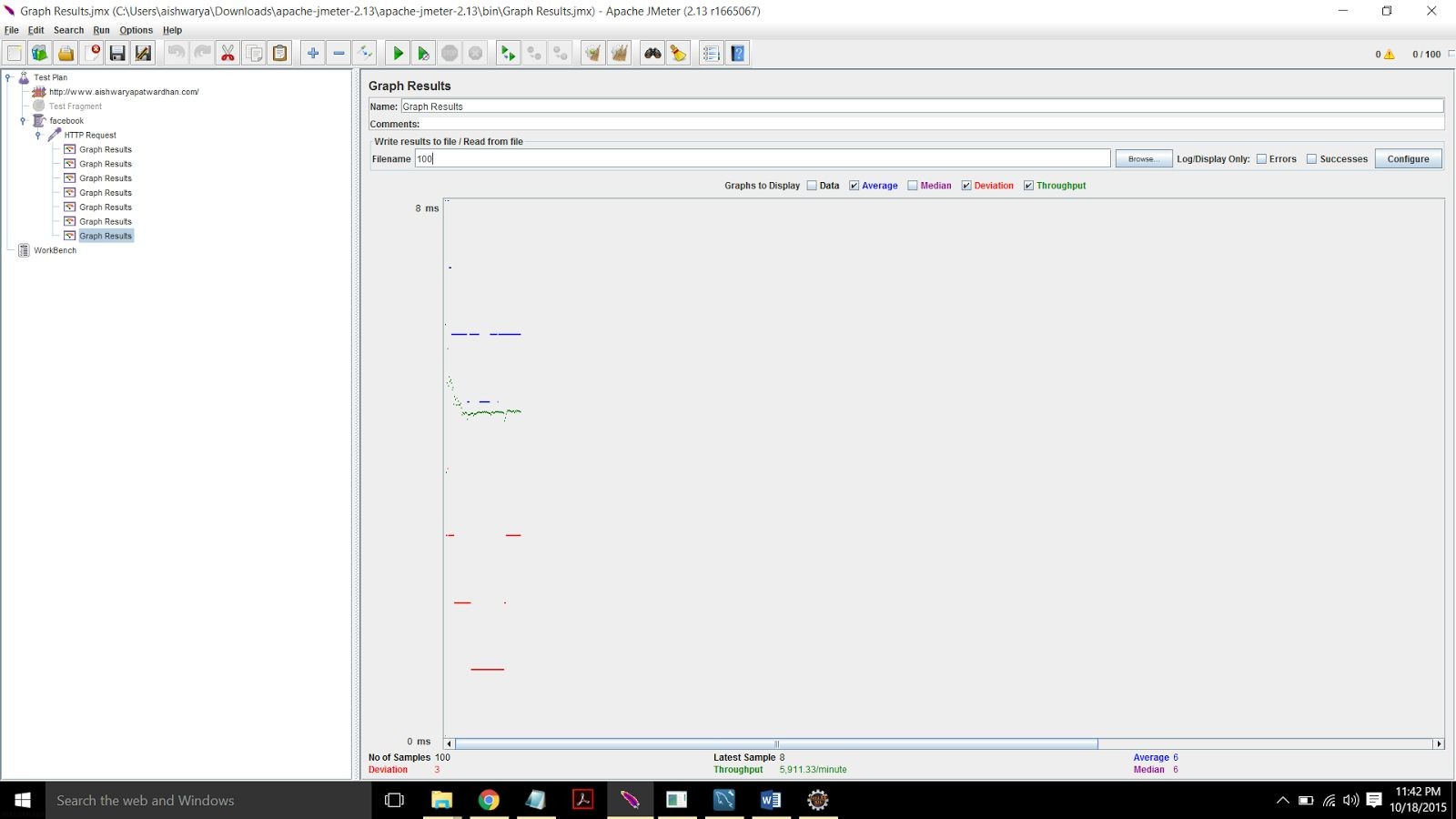
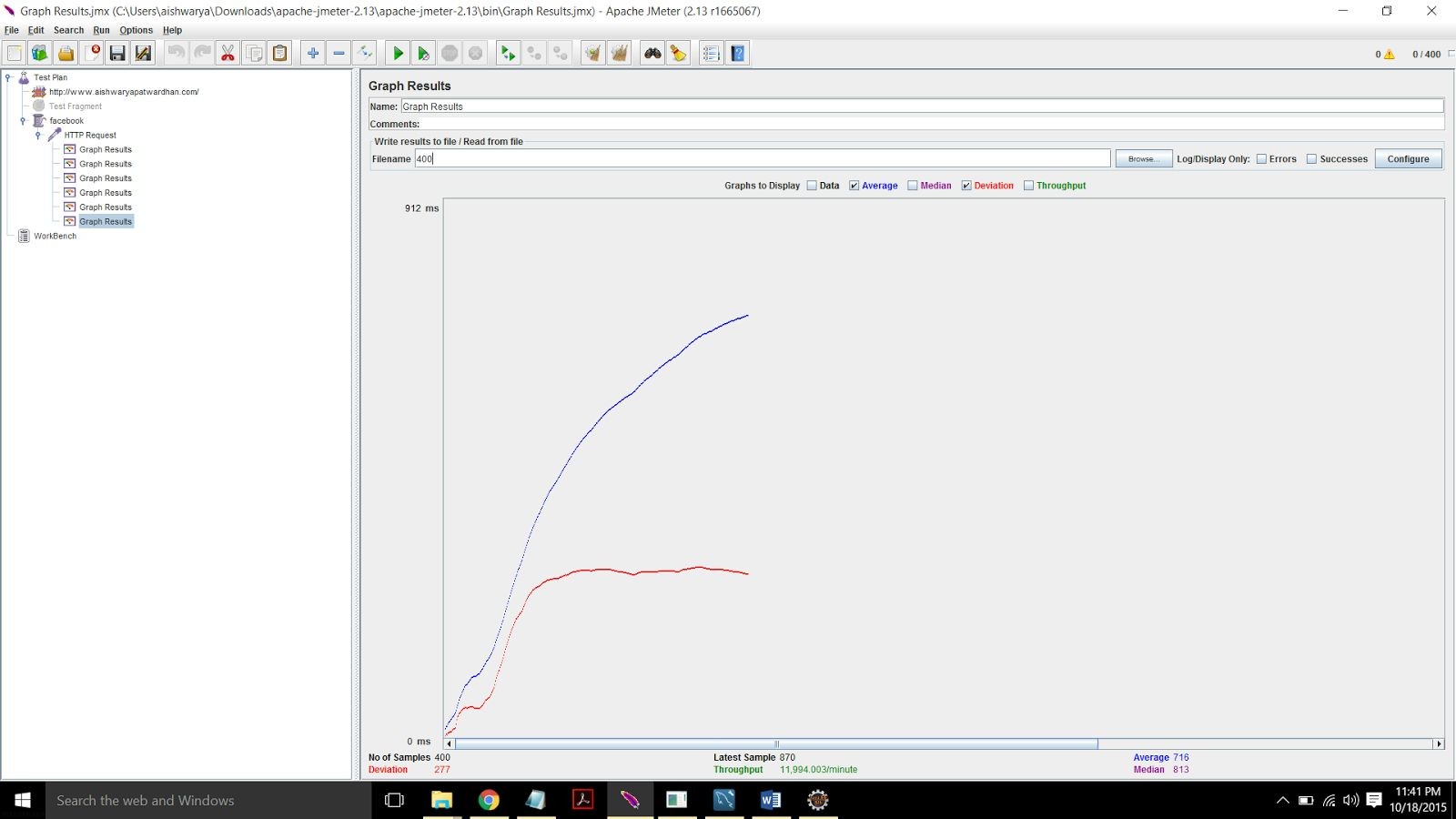
Client 2 – “Facebook Client”

Test client can interact with java script. with

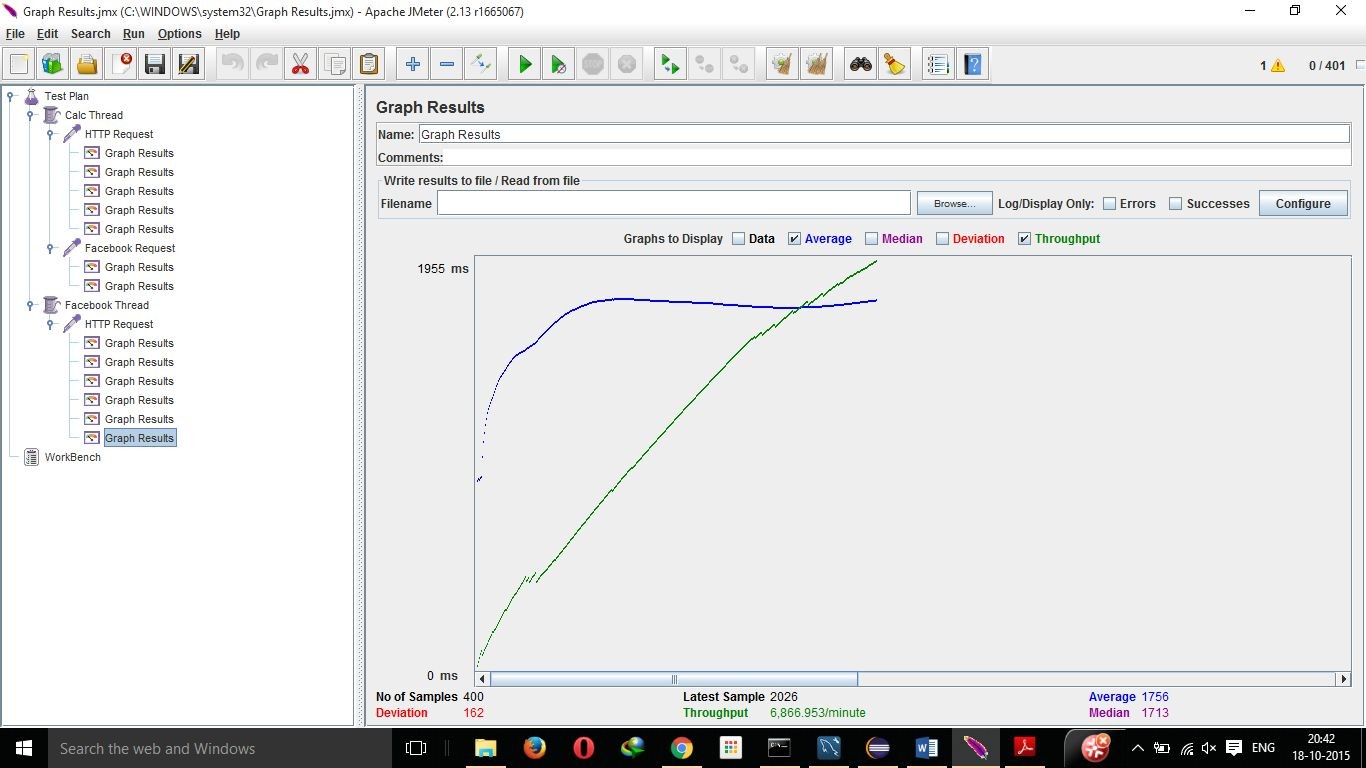
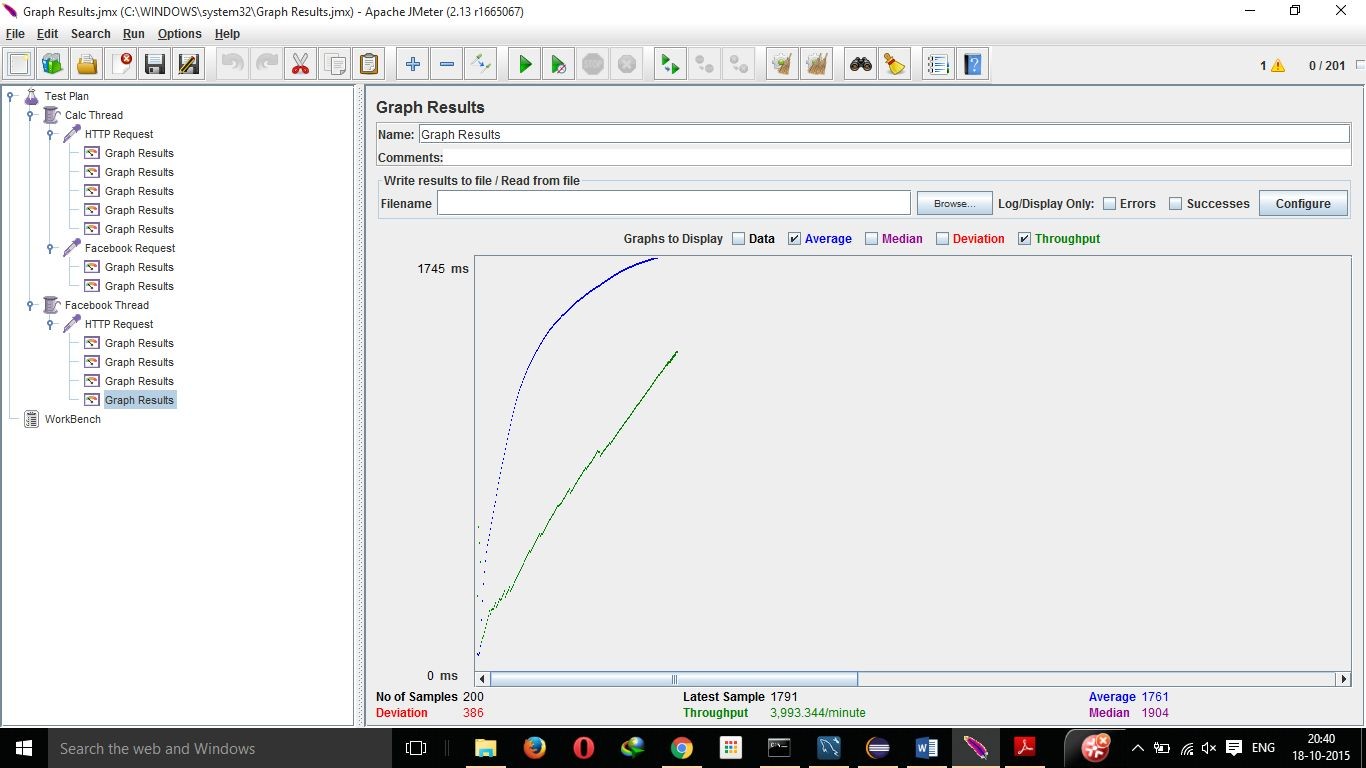
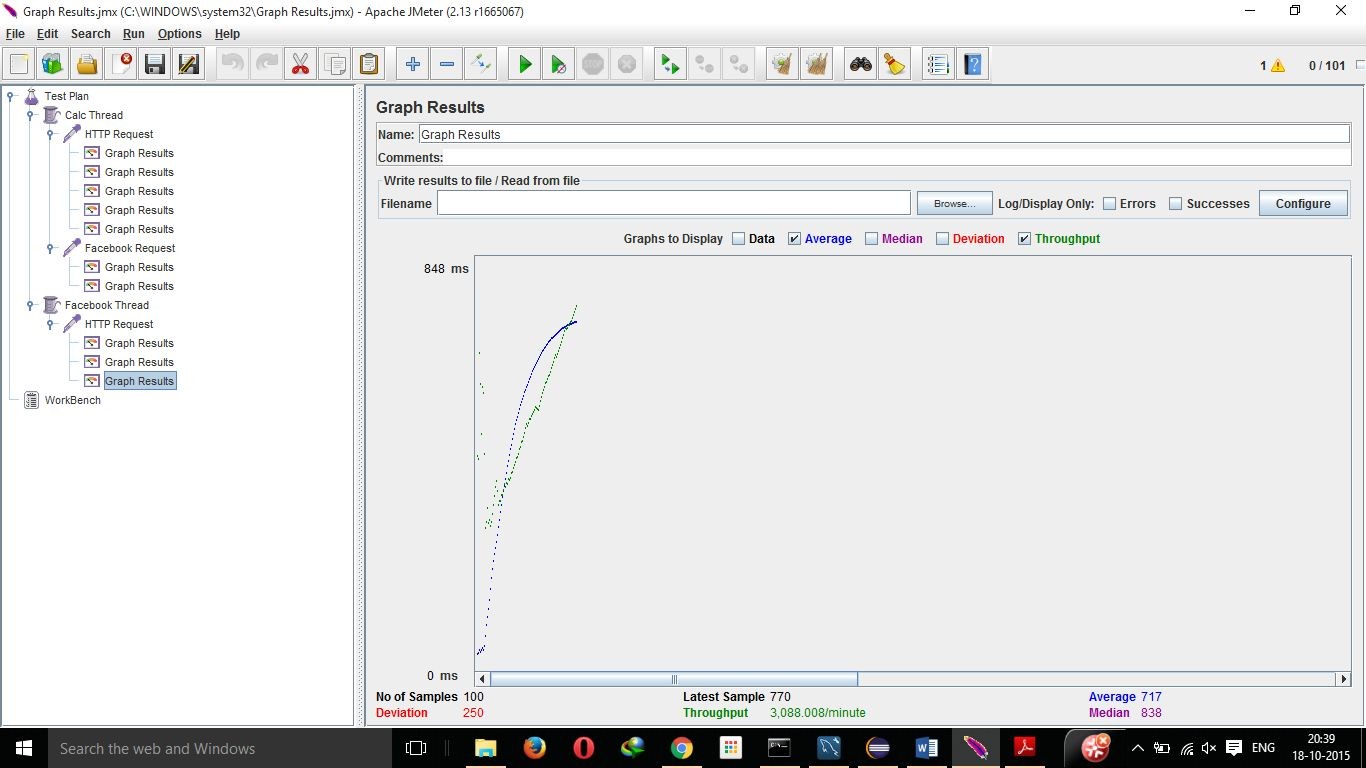
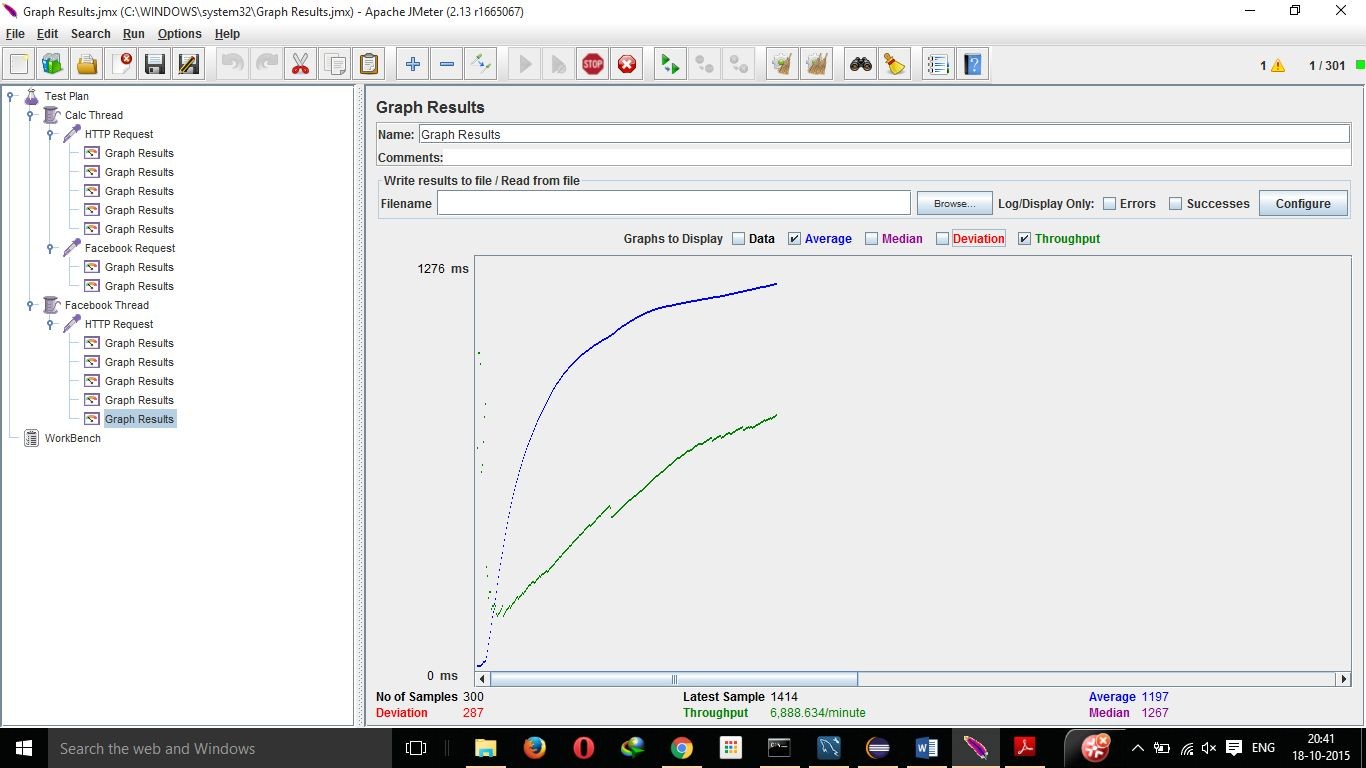
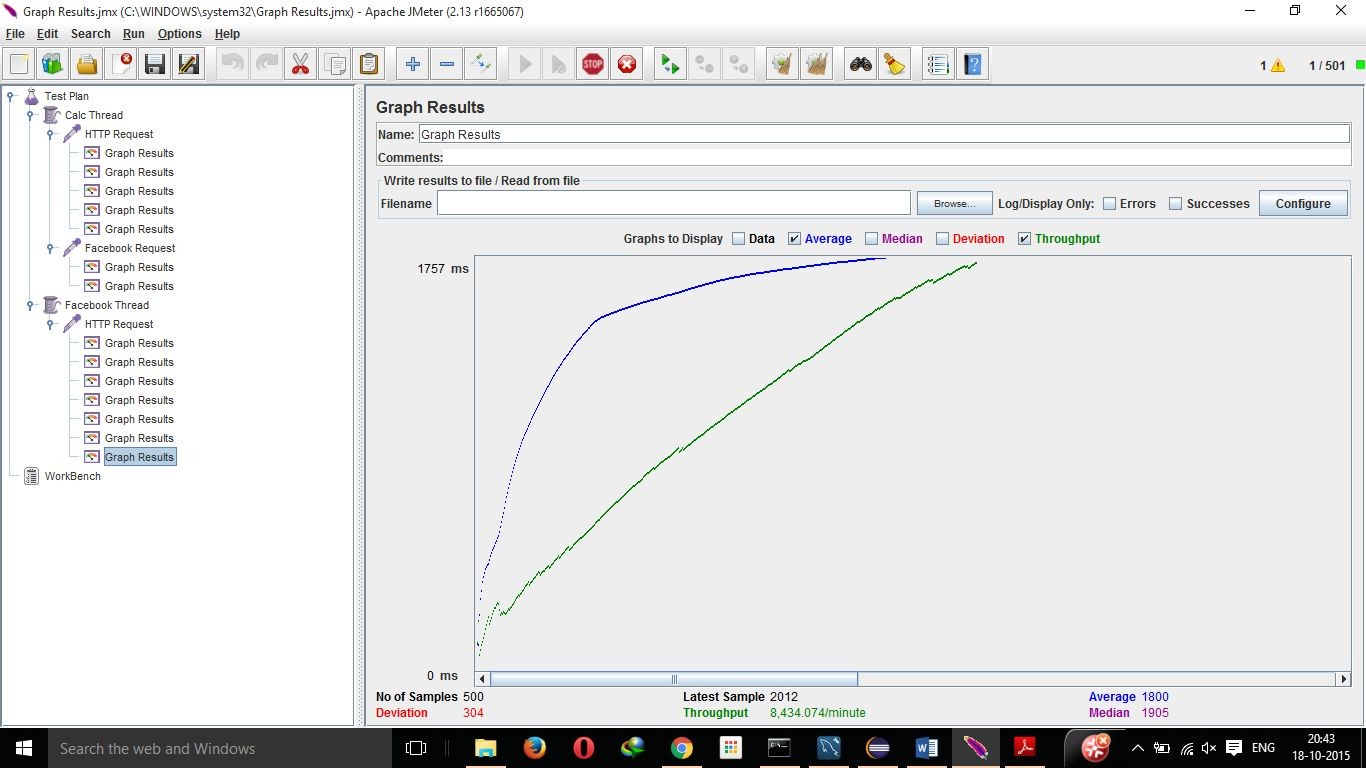
The client should provide the facility to test all the functions of “Facebook application” service up to 500 concurrent users.

Show the graph with 100, 200,300, 400, 500 users without connection pooling and with connection pooling.

Without connection pooling



**iWith connection pooling**



**­­\*\*­­**

1. Describe your algorithm and strategy for data caching. Discuss why you choose the

algorithm/strategy.

For data caching out of all strategy SLRU provides a good result. Algorithm:­

Segmented LRU (SLRU)

An SLRU cache is divided into two segments, a probationary segment and a protected segment. Lines in each segment are ordered from the most to the least recently accessed. Data from misses is added to the cache at the most recently accessed end of the probationary segment. Hits are removed from wherever they currently reside and added to the most recently accessed end of the protected segment. Lines in the protected segment

have thus been accessed at least twice. The protected segment is finite, so migration of a line from the probationary segment to the protected segment may force the migration of the LRU line in the protected segment to the most recently used (MRU) end of the probationary segment, giving this line another chance to be accessed before being replaced. The size limit on the protected segment is an SLRU parameter that varies according to the I/O workload patterns. Whenever data must be discarded from the cache, lines are obtained from the LRU end of the probationary segment.

2. Compare the performance characteristics (draw graphs) of the servers without connection pooling and with connection pooling. Repeat the same with data caching. Use a graph or table to chart your results visually. Explain why you see such results.

To measure the performance of the server , the server of LinkedIn was tested with 100, 200, 300, 400, 500 concurrent clients. As we can see in above graph the average response time for server is less for when there are 100 concurrent clients and connection pooling is used but response time is greater when connection pooling is not used.

In addition, with 200, 300, 400, 500 concurrent client request average response time is less for server without connection pooling. Here in this case this server failed to respond after few requests from clients as number of requests were more as compared to 100 connections therefore it displayed error that server failed too many connections and server with connection pooling continue with response without any failure.

3. Considering data caching, how you would implement Request Caching strategy. Explain in detail. You do not need to implement this new function

SQL request caching is caching the SQL query statement along with its result. Server will have to respond for each query from client but when there are similar queries server will return the result from query cache rather than executing same query again and again. As read and write operations are performed continuously, the query cache will be useful in case when number of read operations are more than write operation therefore operation such as retrieval of data from database will be

performed efficient and faster way . On the other hand, query cache is not suited for insert, update, delete operations.

On modification of data in database, data in query cache are emptied. Steps to implement SQL query cache in MySQL as follows

Add configuration directives to the MySQL server as follows: query\_cache\_size = SIZE (Allocation of memory for caching results). query\_cache\_type = OPTION (With three options)

1. Option 0: No cahing.

2. Option 1: Cache query results except those that start with SELECT S\_NO\_CACHE.

3. Option 2: Cache query results that begin with SELECT SQL\_CACHE. Add the configuration directives to the MySQL server,

query\_cache\_size = 50;

query\_cache\_type = 2;

Example:­"SELECT SQL\_CACHE \* FROM tabletype;"

4. Is your session strategy horizontally scalable? If YES, explain your session handling strategy. If NO, then explain how can you achieve it.

No ,to achieve horizontal scalability

**Horizontal Scaling Best Practices**

**Split Your Monolithic Application**

The idea behind this practice is to split the monolithic application into groups of functionally related services which can be maintained and scaled together. You can do this via SOA(Service Oriented Architecture), ROA(Resource Oriented Architecture), or by just following good design practices, idea is just to split big monolithic application into smaller applications based on functionality. For example, all user related services can be grouped into one set, search in another, etc. Web scalability is about developing loosely coupled systems. It should be designed in such a way that many independent components communicate with each other. If one component goes down, it should not effect the entire system. This help avoids “single points of failure”. The more decoupled unrelated functionality can be, the more flexibility you will have to scale them independently of one another. As services are now split, the actions we can perform and the code necessary to perform them are split up as well. This means that different teams can become experts in subsets of systems and don’t need to worry about other parts of system. This not only helps in scaling application tier but helps in scaling database tier as well. As rather using single database and going with one choice , you can choose different databases for different needs.

**Use Distributed Caching**

Distributed caching can help in horizontal scalability of a web application by avoiding access to a slow database or filesystem and instead retrieves data directly from the fast local memory. This helps the application in scaling linearly, just by adding more nodes to the cache cluster. A Java web application using a distributed cache can store frequently accessed data such as results of a database query or computation intensive work in a cache. Applications can use Memcached or Infinispan to create distributed cache cluster. Caching is all about minimizing the amount of work a

system does. It is advisable that you put caching in its own tier rather than using application servers machines. This will help you in scaling the caching tier independently of application tier.

**Use CDN**

You should use CDN(Content delivery network) to offload traffic from your web application. A content delivery network or content distribution network (CDN) is a large distributed system of servers deployed in multiple data centers across the Internet. The goal of a CDN is to serve content to end­users with high availability and high performance. CDNs are mostly used for delivering static content like css , images , javascript, static html pages near to the user location. It will find the best possible server which can fulfill the request in the least amount of time by fewer network hops, highest availability, or fewer request. Your application can leverage either Akamai or Amazon CloudFrond for CDN capabilities.

**Deploy Shared Services To Their Own Cluster**

Some applications use file systems to save files uploaded by users, or to store configuration files. You need to replicate these files to all the nodes so that all nodes can use them. With more nodes added, copying files among server instances will occupy all the network bandwidth and consuming considerable CPU resources. To work in a cluster, the solution is to use the database in place of external files, or SAN or use Amazon S3. This will help achieve better scalability.

**Go Async**

The next best practice to scaling is the use of asynchronous calls. If two components X and Y call each other synchronously, then X and Y are tightly coupled, and then either both of them will scale or none of X and Y will scale. This is a characteristic of tightly coupled systems – to scale X, you must scale Y as well and vice versa. For example, it is very common that after user registration email is sent to the user for verification. Now if you tightly couple the user registration service and email service together than scalability of user registration service will be dependent on tje email service. But if you do it asyncronously either through a queue, multicast messaging, or some other

means, then you can continue registering users, untill you are sure that the verification email will be sent to user. Synchronous calls stop the entire program execution waiting for a response, which ties all services together leading to cascading failures. This not only impacts scalability but availability of the system as well. In other words, if Y is down then X is down. With async design, X and Y now have independent availability characteristics – X can continue to move forward even if Y is down.

**Parallelize The Task**

There are times when you can divide a single threaded task to multiple smaller tasks which can be run in parallel not only on a single machine but on a cluster of machine. A single thread of tasks will be the scalability bottleneck of the system. Java 7 introduced fork/join framework that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of

your application.

**Don’t Store State in the Application Tier**

The golden rule to achieve scalability is not storing state in the application tier but storing state in the database so that each node in the cluster can access the state. Then you can use a standard load­balancer to route incoming traffic. Because all application servers are equal and does not have any transactional state, any of them will be able to process the request. If we need more processing power, we simply add more application servers.

**Use Non­Blocking IO**

The java.nio package allows developers to achieve greater performance in data processing and offers better scalability. The non­blocking I/O operations provided by NIO and NIO.2 boosts Java application performance by getting “closer to the metal” of a Java program, meaning that the NIO

and NIO.2 APIs expose lower­level­system operating­system (OS) entry points. In a web application, traditional blocking I/O will use a dedicated working thread for every incoming request. The assigned thread will be responsible for the whole life cycle of the request – reading the request data from the

network, decoding the parameters, computing or calling other business logical functions, encoding the result, and sending it out to the requester. Then this thread will return to the thread pool and be reused by other requests. With NIO, multiple HTTP connections can be handled by a single thread and the limit is dependent on amount of heap memory available. You can turn on NIO in Tomcat by changing the protocol attribute of <Connector> element as shown below

<Connector protocol="org.apache.coyote.http11.Http11NioProtocol" port="80"

redirectPort="8443" connectionTimeout="20000" compression="on" />

5. Explain how node.js handles multiple client requests simultaneously even though it is single threaded. Explain using block diagram.

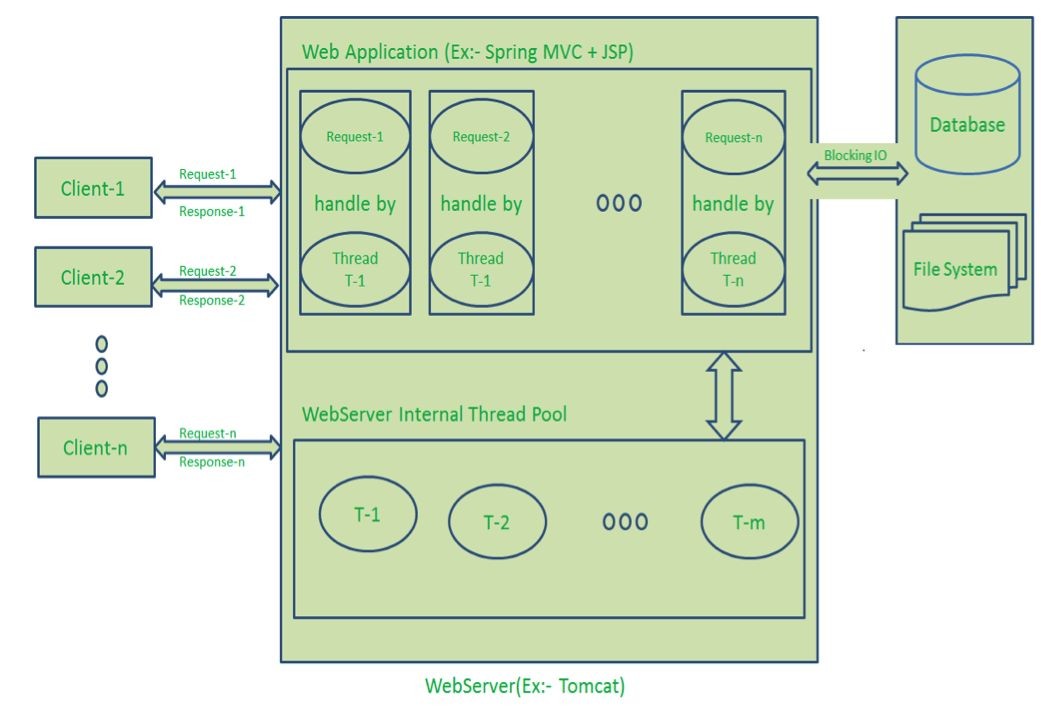


Diagram Description:

● Here “n” number of Clients Send request to Web Server. Let us assume they are accessing our Web Application concurrently.

● Let us assume, our Clients are Client­1, Client­2… and Client­n.

● Web Server internally maintains a Limited Thread pool. Let us assume “m” number of Threads in Thread pool.

● Node JS Web Server receives Client­1, Client­2… and Client­n Requests and places them in the Event Queue.

● Node JS Even Loop Picks up those requests one by one.

○ Even Loop pickups Client­1 Request­1

■ Checks whether Client­1 Request­1 does require any Blocking IO Operations or takes more time for complex computation tasks.

■ As this request is simple computation and Non­Blocking IO

task, it does not require separate Thread to process it.

■ Event Loop process all steps provided in that Client­1

Request­1 Operation (Here Operations means Java Script’s functions) and prepares Response­1

■ Event Loop sends Response­1 to Client­1

○ Even Loop pickups Client­2 Request­2

■ Checks whether Client­2 Request­2does require any Blocking IO Operations or takes more time for complex computation tasks.

■ As this request is simple computation and Non­Blocking IO

task, it does not require separate Thread to process it.

■ Event Loop process all steps provided in that Client­2

Request­2 Operation and prepares Response­2

■ Event Loop sends Response­2 to Client­2

○ Even Loop pickups Client­n Request­n

■ Checks whether Client­n Request­n does require any Blocking IO Operations or takes more time for complex computation tasks.

■ As this request is very complex computation or Blocking IO

task, Even Loop does not process this request.

■ Event Loop picks up Thread T­1 from Internal Thread pool and assigns this Client­n Request­n to Thread T­1

■ Thread T­1 reads and process Request­n, perform necessary Blocking IO or Computation task, and finally prepares Response­n

■ Thread T­1 sends this Response­n to Event Loop

■ Event Loop in turn, sends this Response­n to Client­n

ref:http://www.journaldev.com/7462/node­js­processing­model­single­threa ded­model­with­event­loop­architecture