**Introduction**

In this lab, development of server-1/client-1 (Calculator application) and server-2/client-2 (Dropbox Application), we are using ReactJS with redux for the client and node.js and express based servers. The goal is to make both the application modular, well structured (so that it can be maintained easily) with user-friendly UI. Also, making the interaction with the database faster using connection pooling so that the user can have a smooth experience while navigating through the application. This is achieved using ReactJS where only components are refreshed on a change of data.

In the calculator application, the client performs API calls to the server where the calculations are performed. The calculator is tested for upto 1000 concurrent calls for 100 concurrent users. Server is also designed to perform exception handling.

In Dropbox prototype application, user is provided with all the basic functionalities of dropbox like uploading/downloading files, file sharing, starring a file for quick access, etc. The UI is made closely similar to Dropbox.com to create a good prototype. Dropbox prototype is stressed tested for upto 500 concurrent users and has performed well.

The design and results of both the applications are explained below in the report.

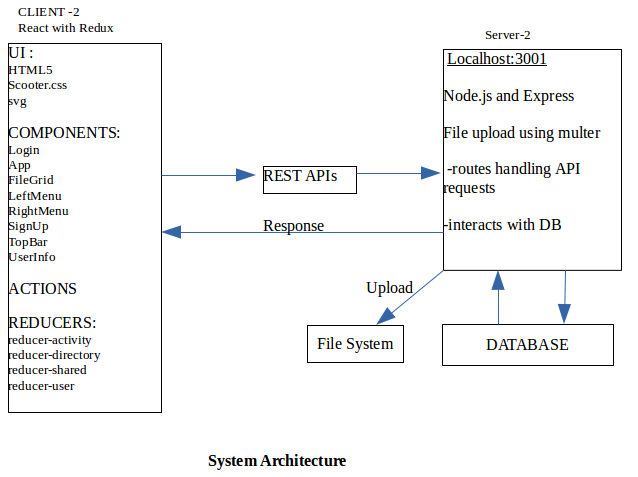
**System Design**

**Server-1/Client-1**

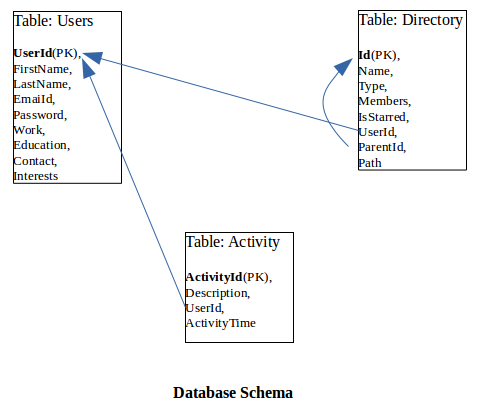
* Server-1 :
  + Node.js and Express based server.
  + Runs on localhost:3001
  + Handles the request from Client-1 for calculation.
  + Capable of performing following tasks: Addition, Multiplication, Subtraction, Division, Exception Handling
* Client-1 :
  + Built using ReactJS and designed using HTML and bootstrap.
  + Located on localhost, port number 3000.
  + User inputs the numbers on Client and sends the inputdata to the server through an API call on the localhost:3001/calculate/doCalculation.

**Server-2/Client-2**

* System design for Dropbox prototype is as shown below in diagram.



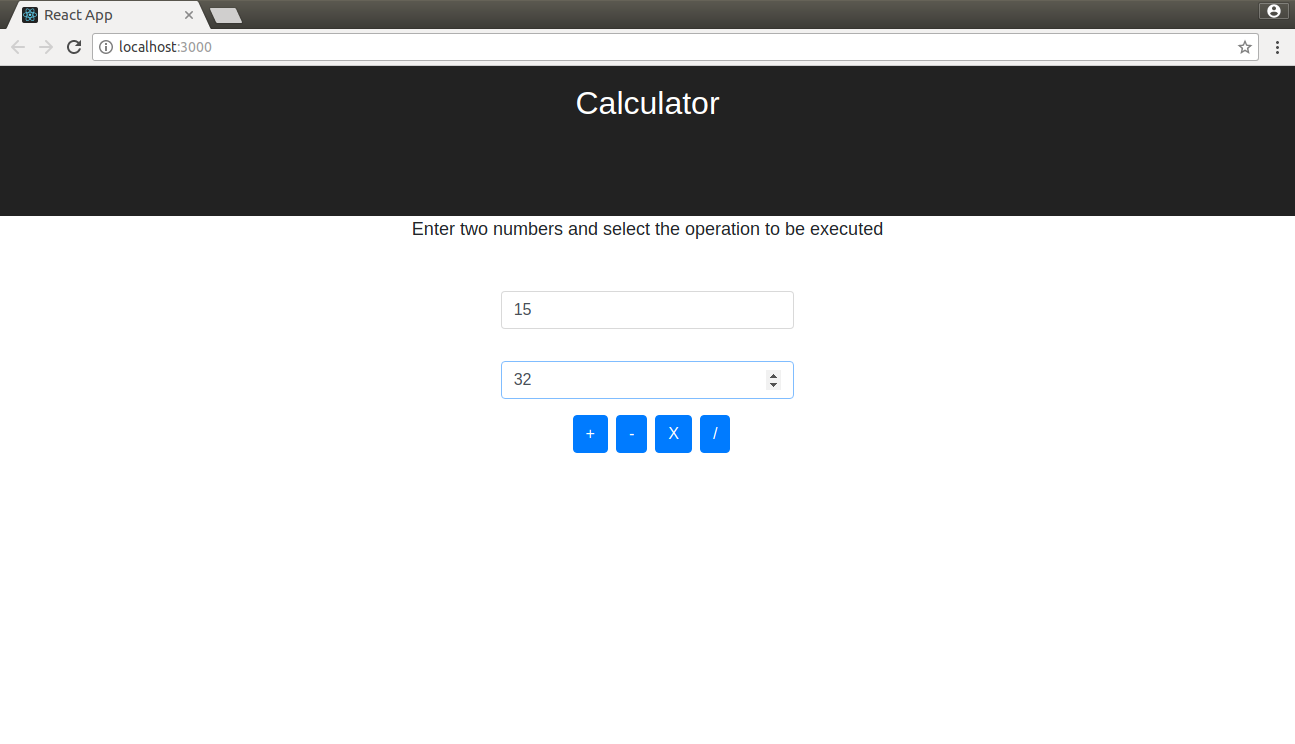
* User interects with client with UI. Client sends the requests for login, signup, DB requests using the REST APIs to the server.
* Server interacts with the and updates Database and also serves the requests for file operations like upload and download.
* File Upload is implemented using the middleware **Multer.**
* File Download is implemented using the react-file-download module of the React.



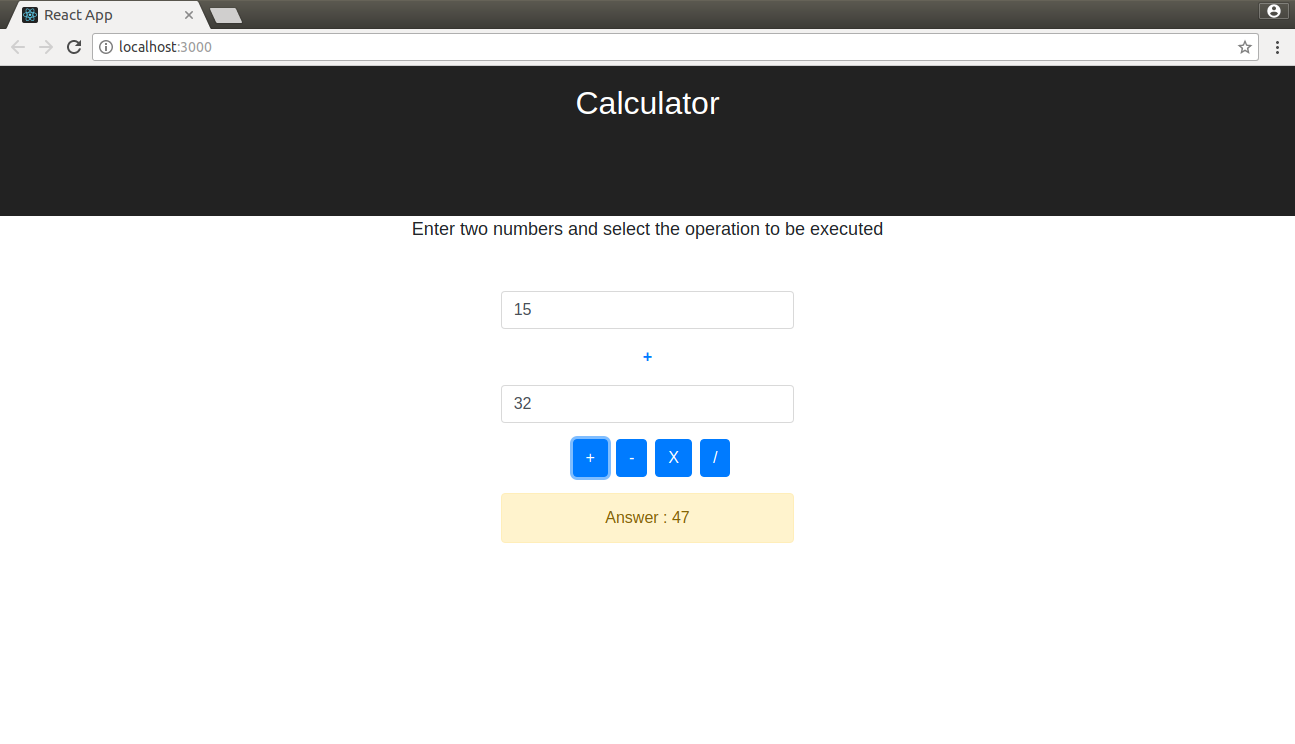
**Results**

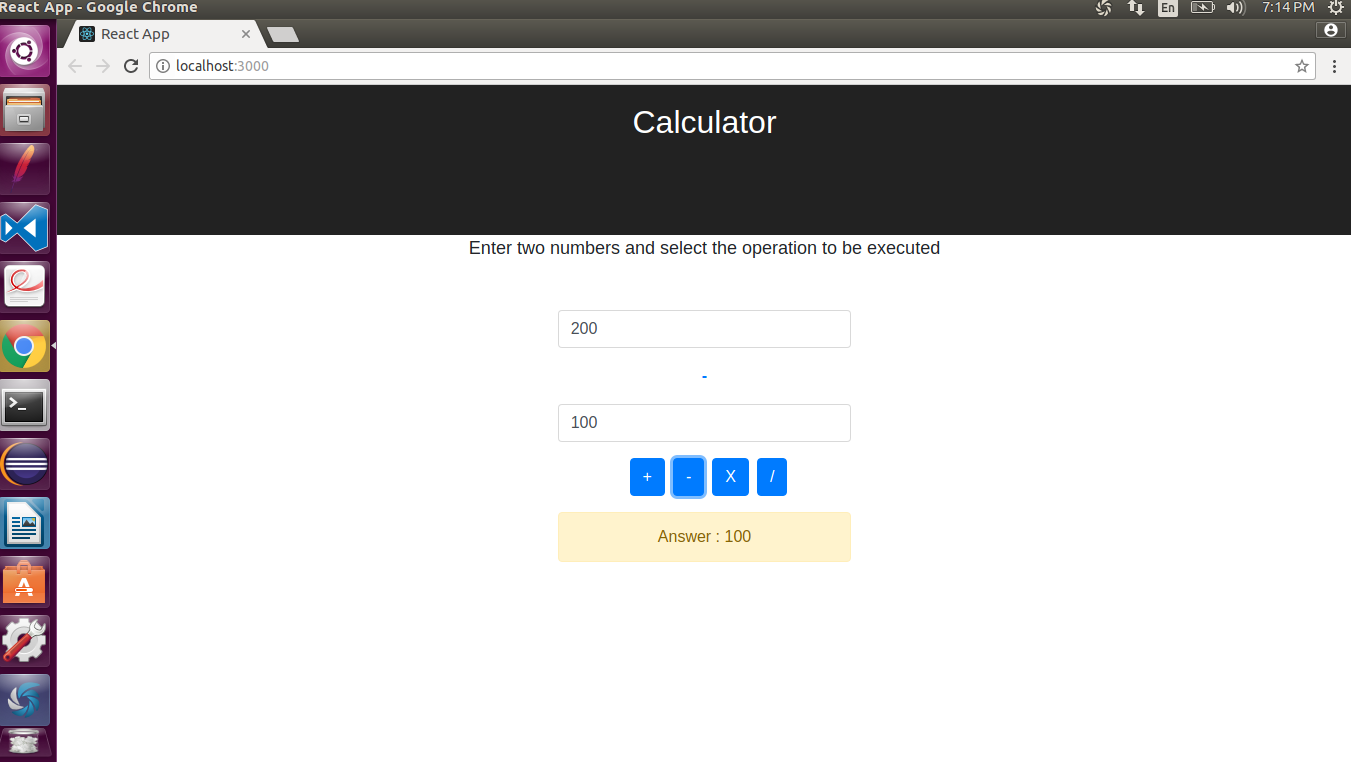
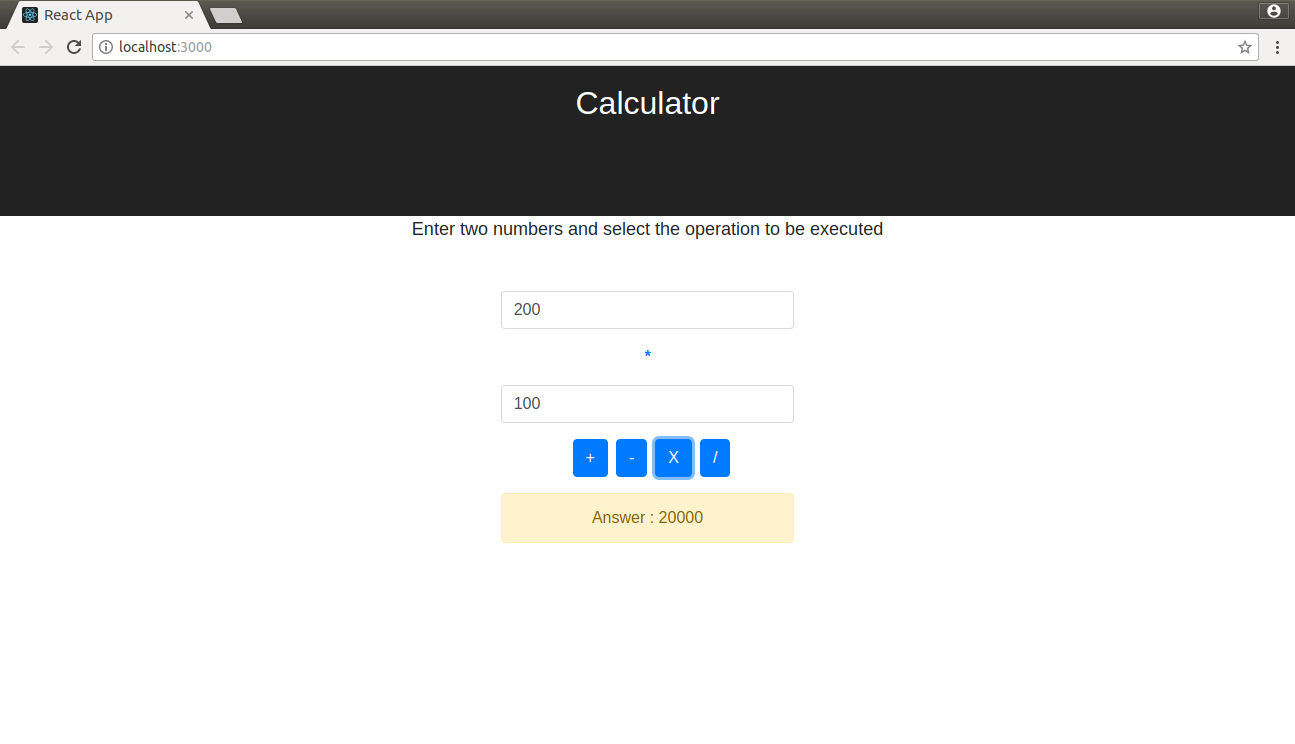
**Calculator:**

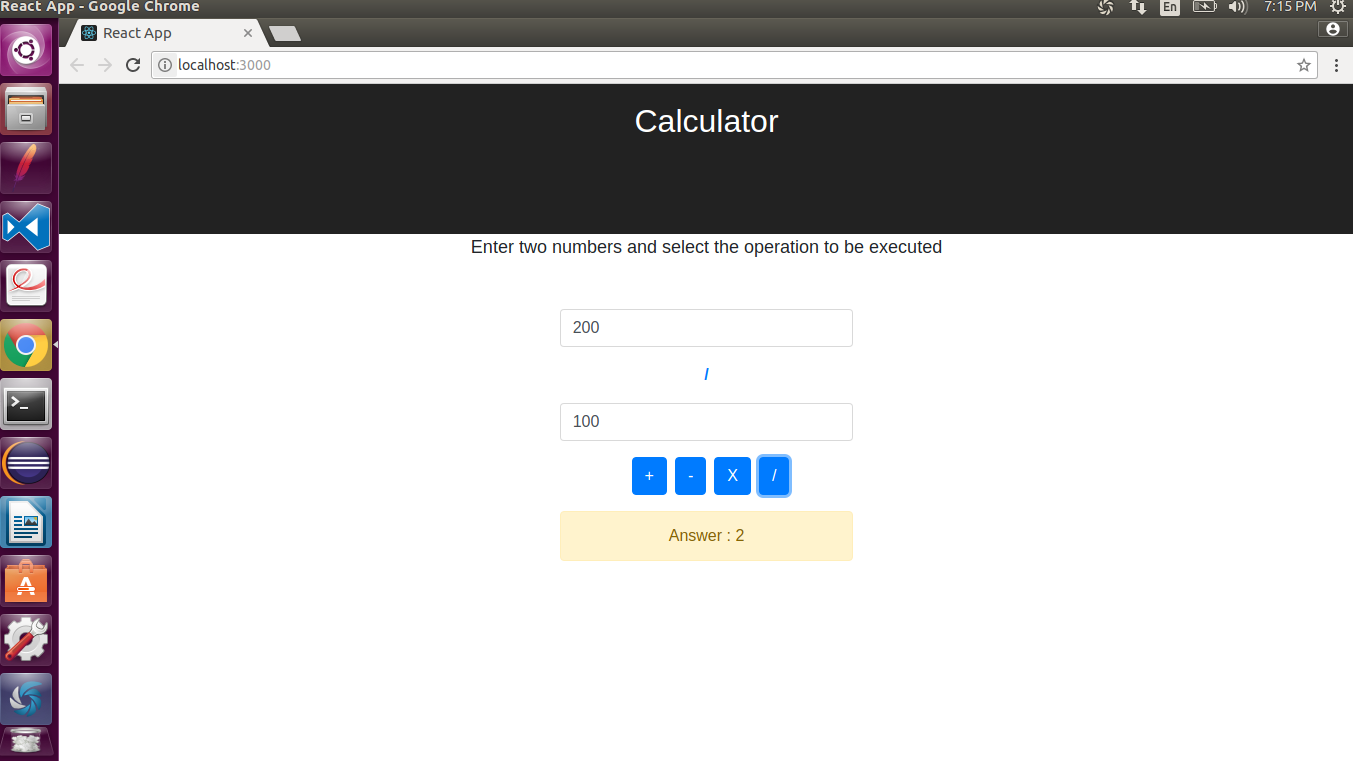
* Homescreen:

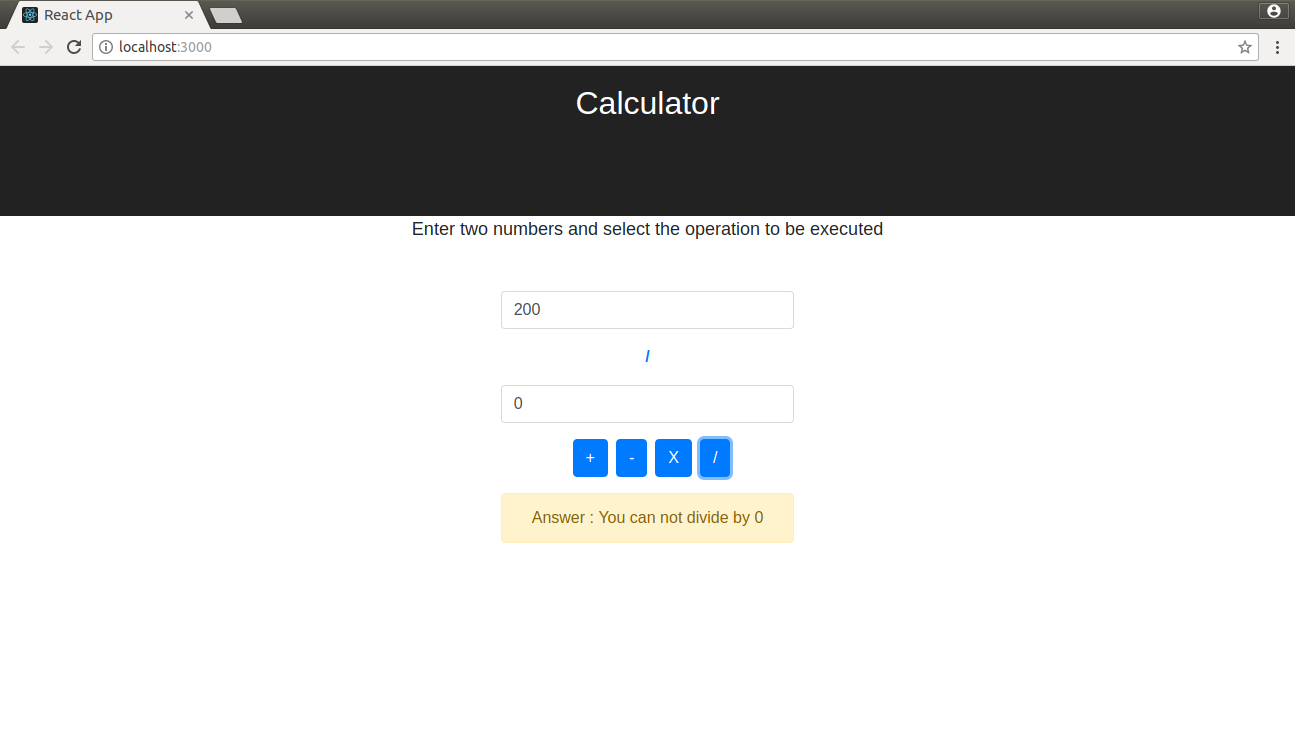


* Addition:

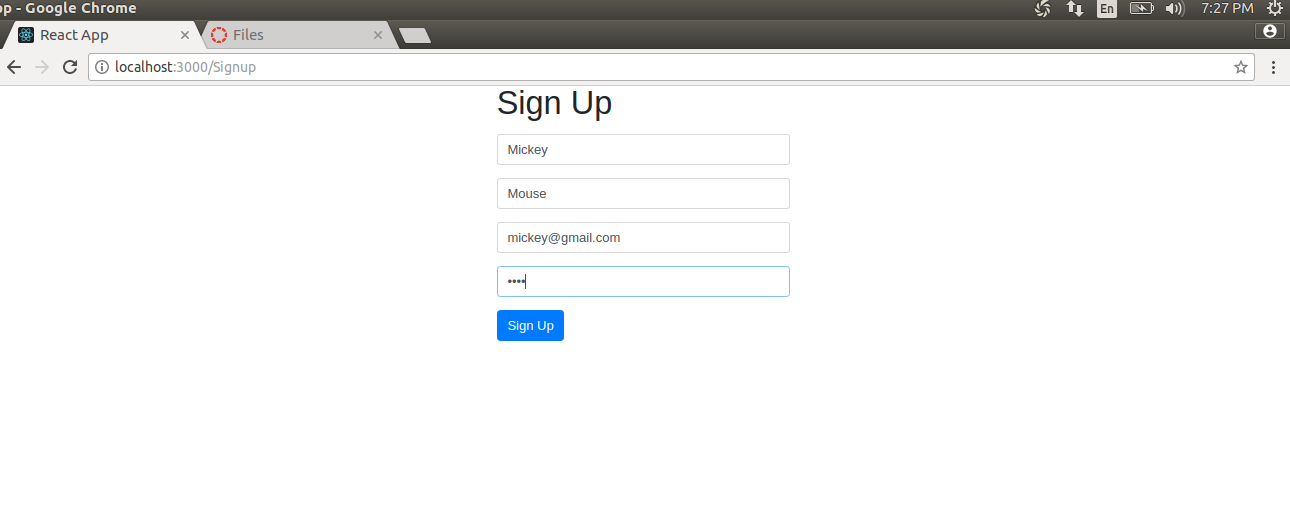


* Subtraction:
* Multiplication:

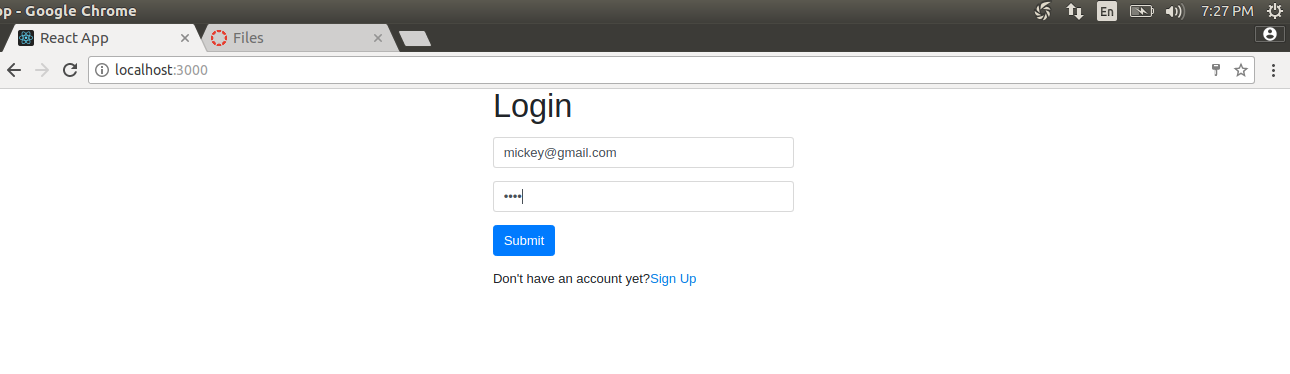
Division:

Exception Handling:

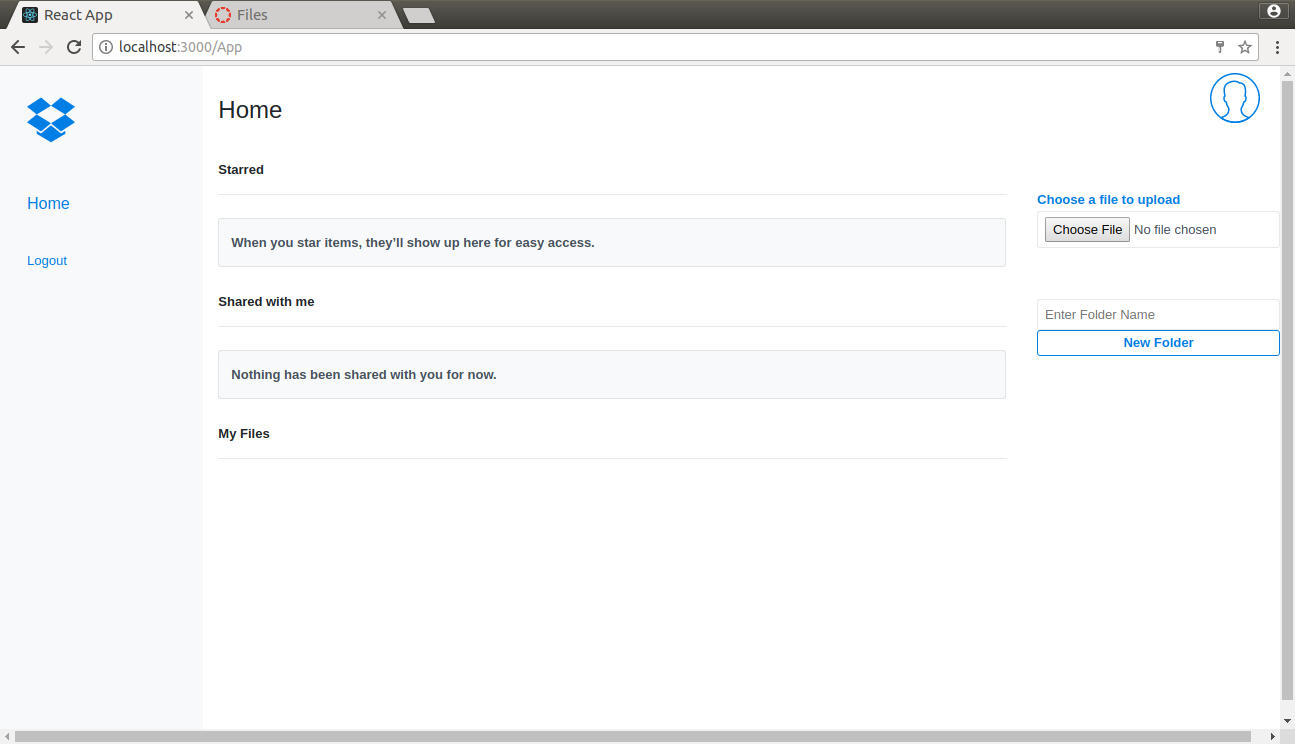
**Dropbox:**

Sign Up new user:

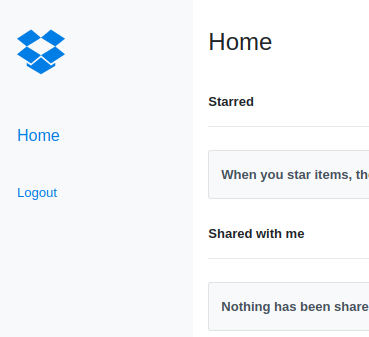
Sign In existing user:



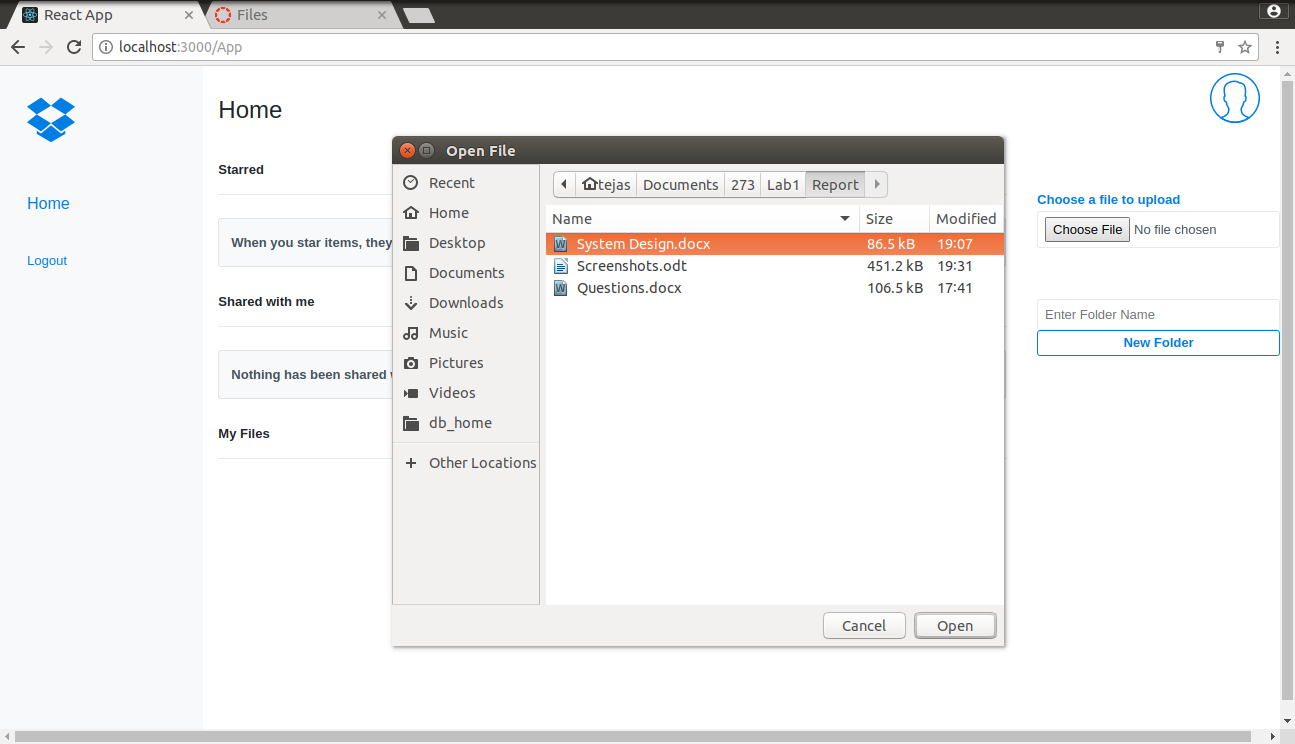
Homescreen:



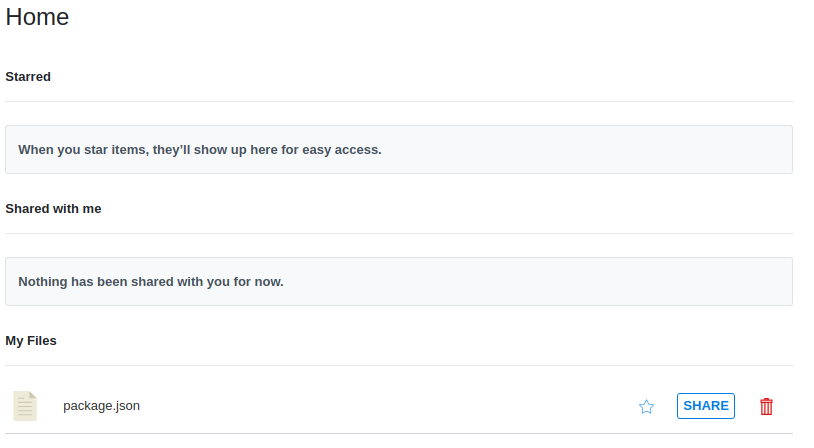
Sign Out:



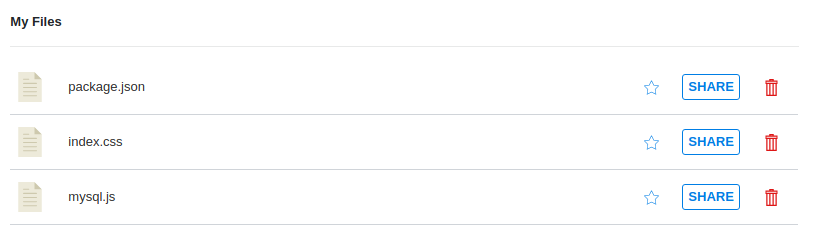
Upload a file:



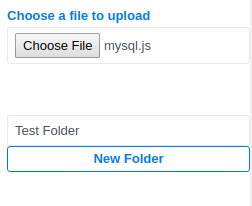
Uploaded file:

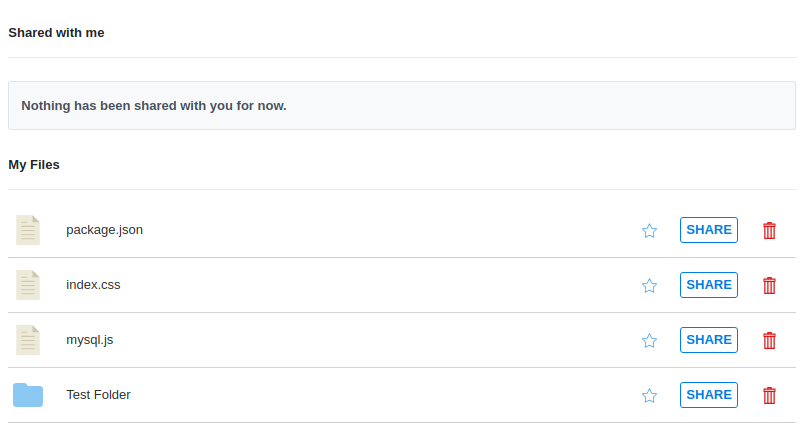


List a file:

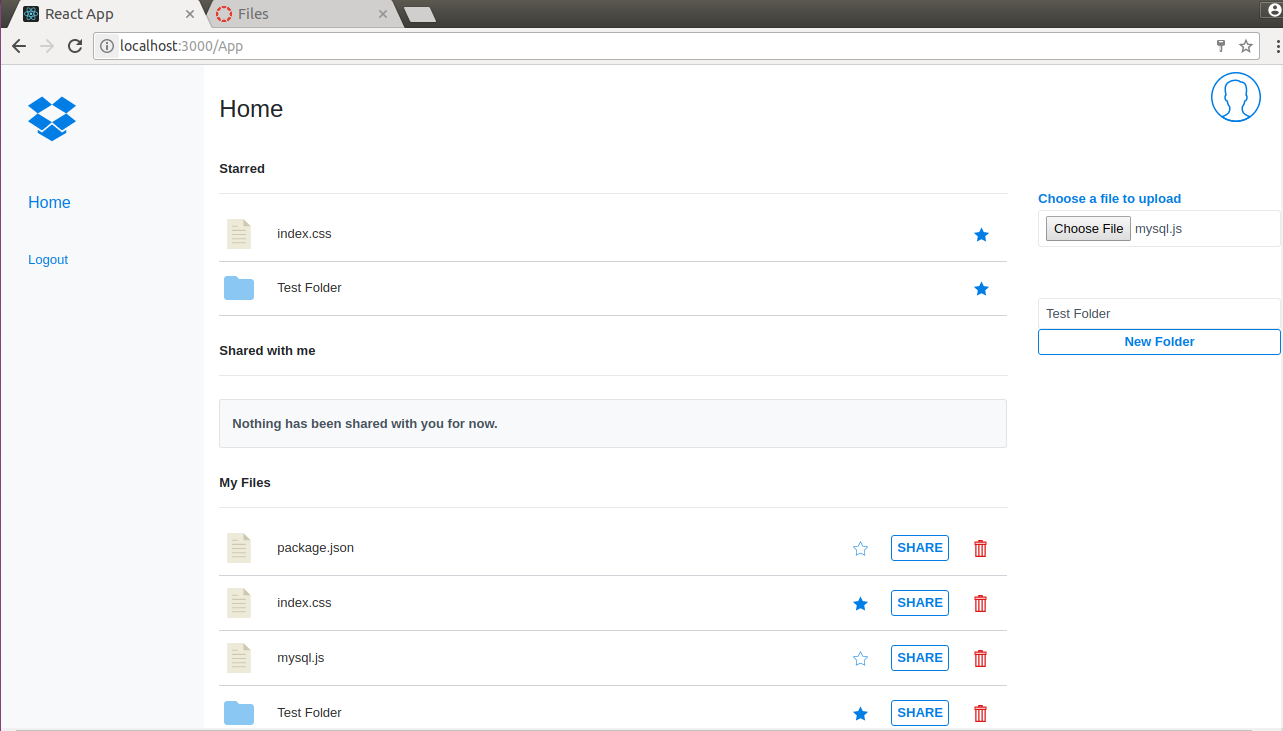


Create a directory:



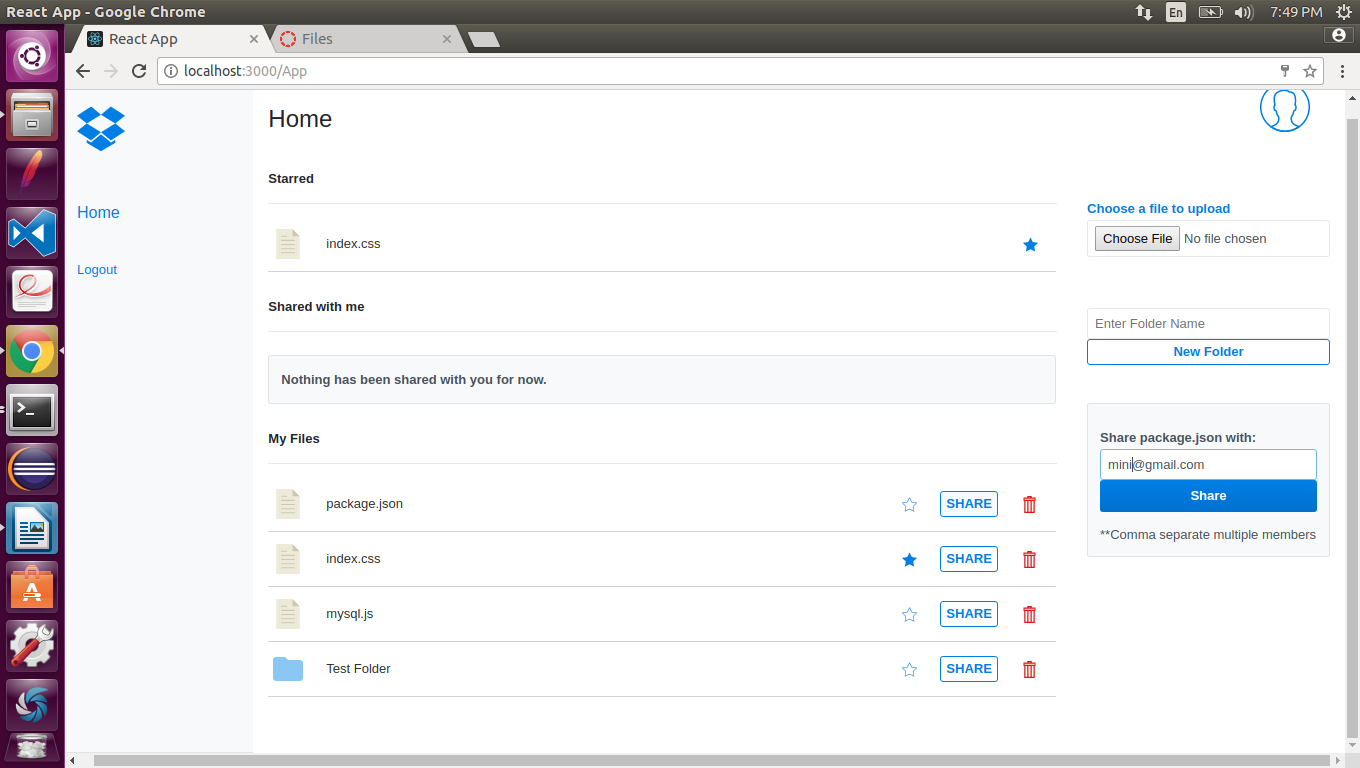


Star a directory/file:

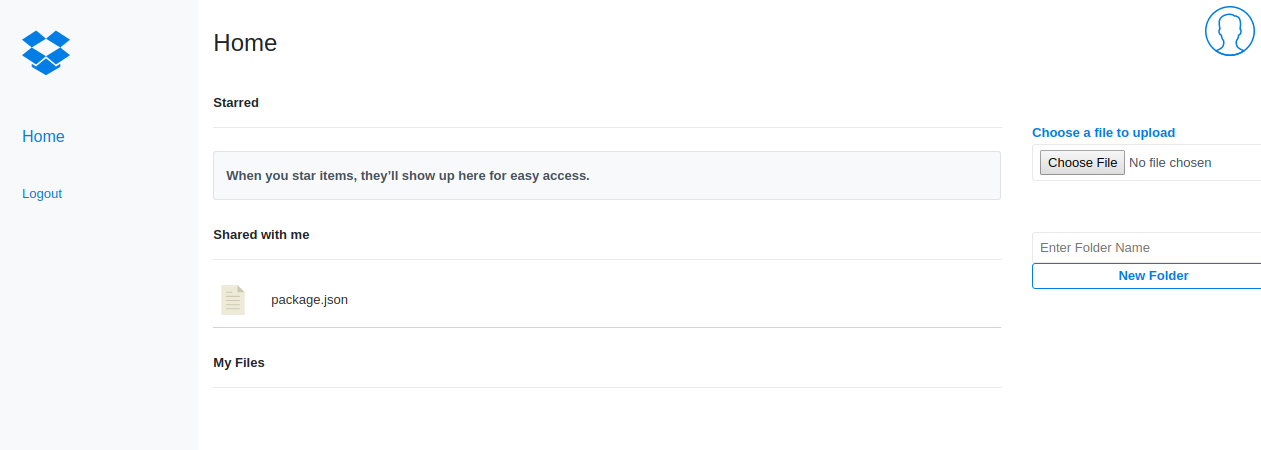


Share a file:

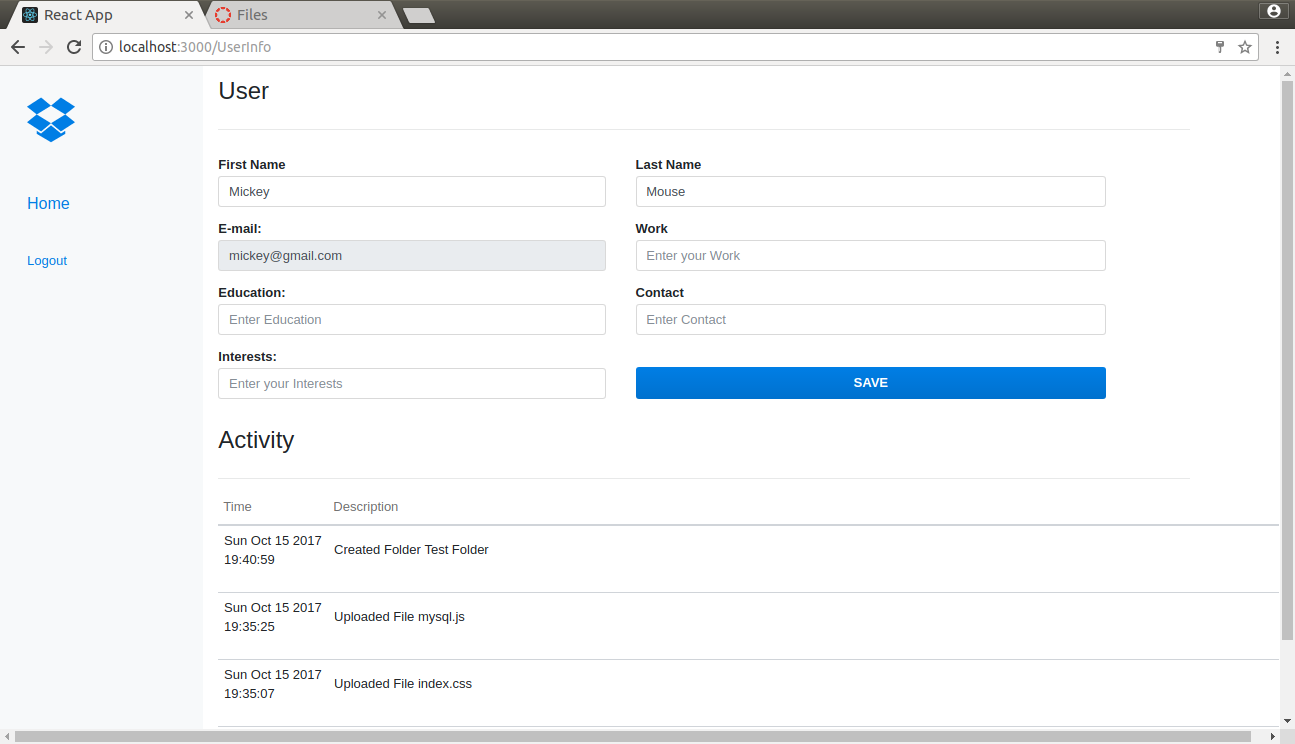
* Sharing package.json with [mickey@gmail.com](mailto:mickey@gmail.com)



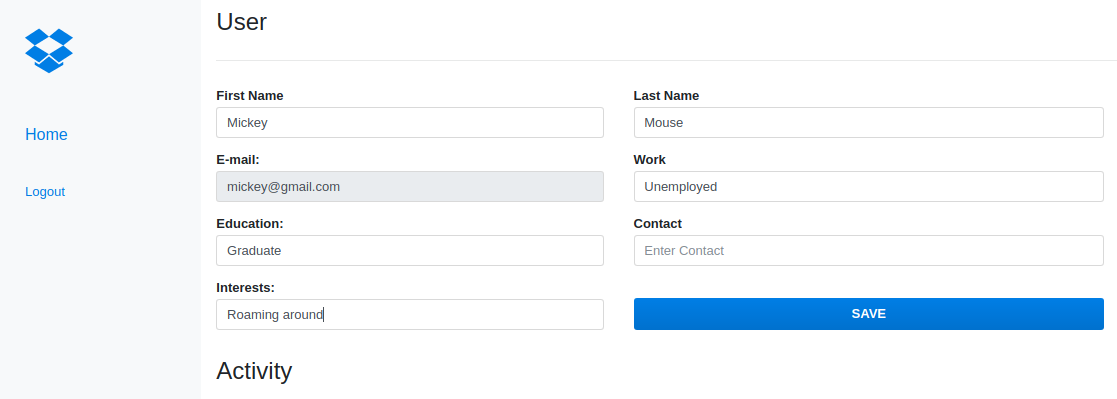
2) Package.json is visible in mini under “shared with me” tab



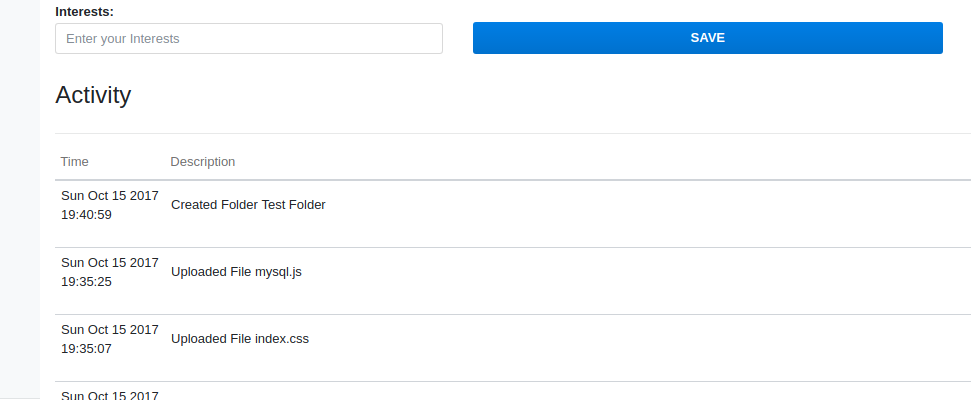
User Detail (On clicking user icon on the upper right corner):



Editable user information:

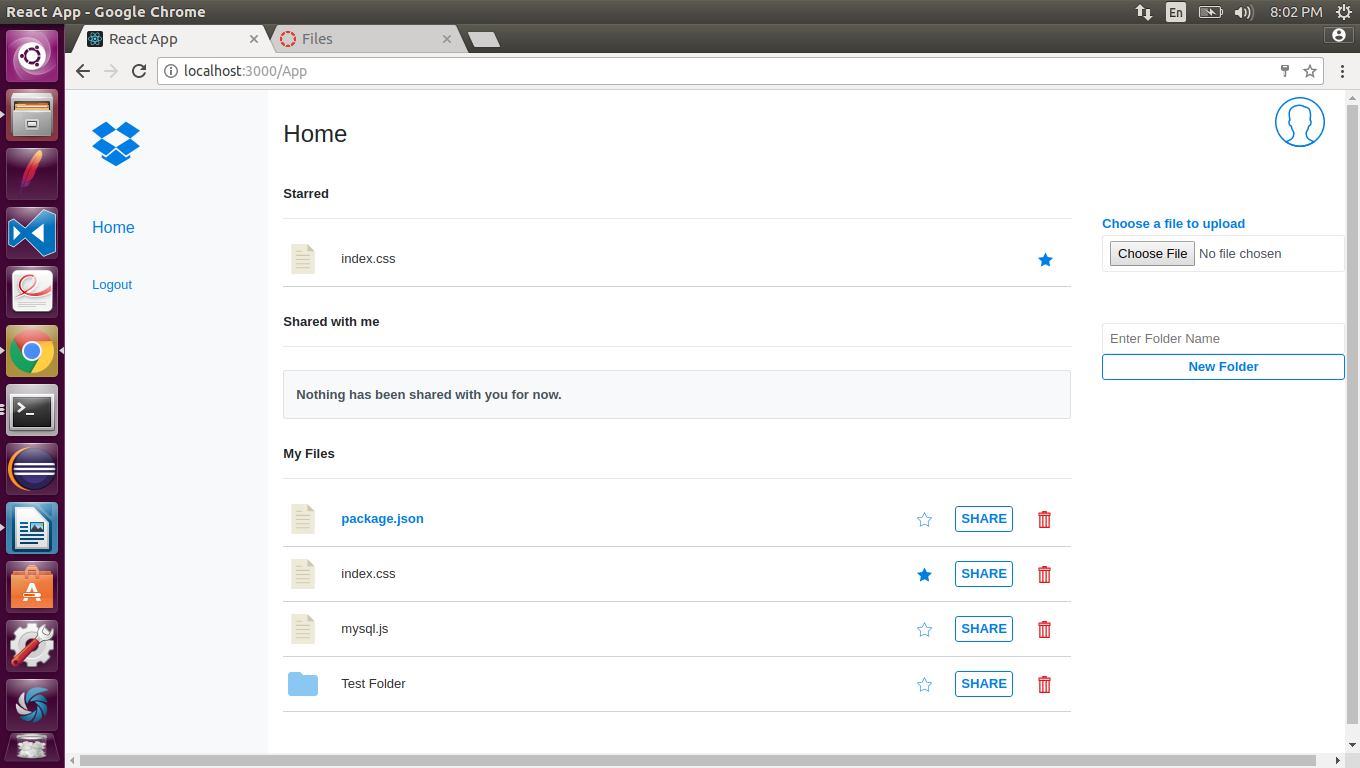


Activity:

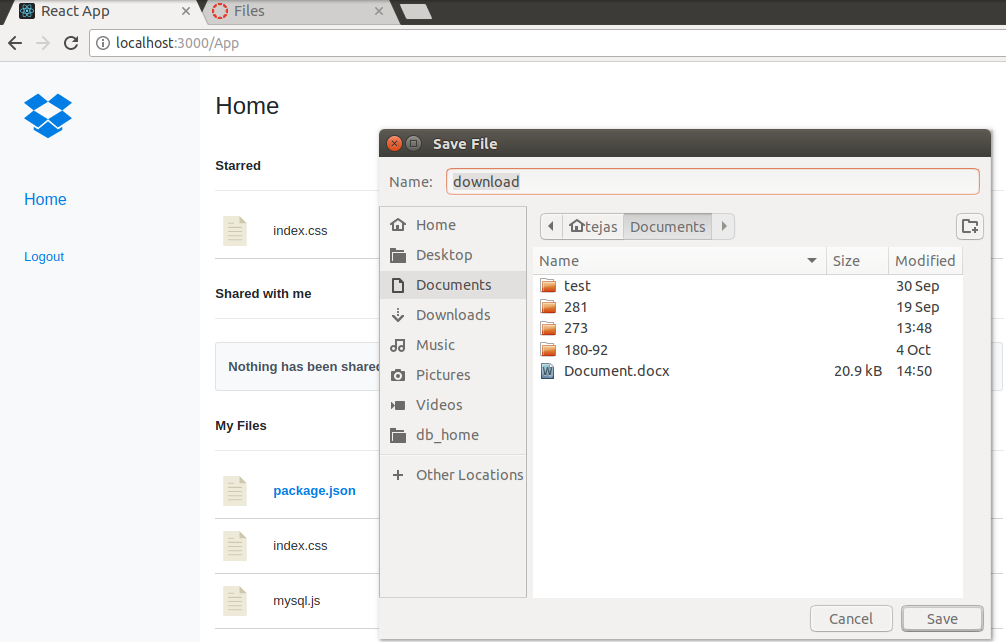


File download:

* file can be downloaded by clicking on file name (filename turns blue on hover):

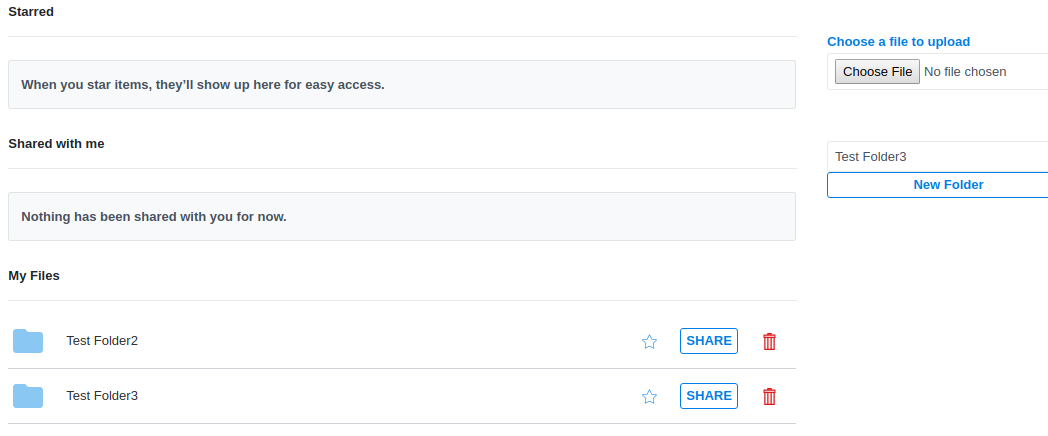


* Clicking on file opens a dialogue box to download file at desired location:

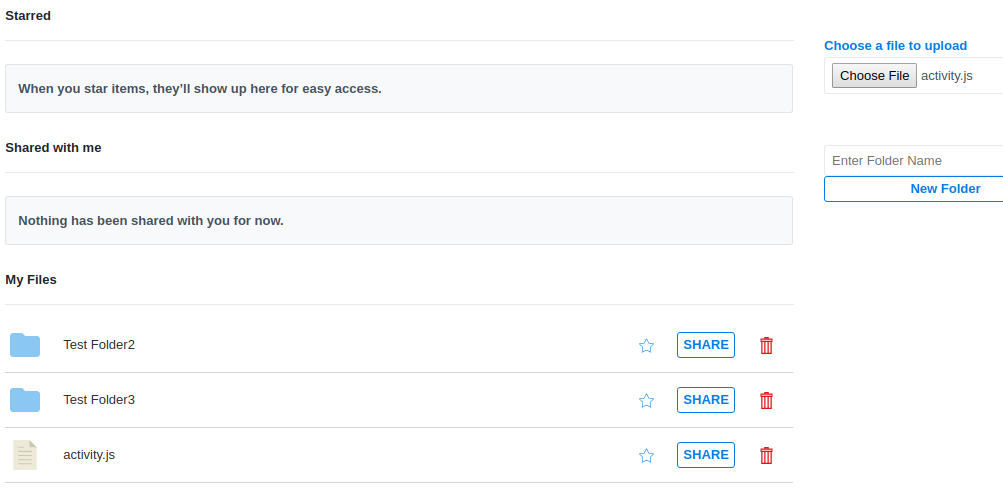


New Folders can be created within a folder (folders can be opened by clicking on the name):

* Here ‘Test Folder2’ and ‘Test Folder3’ are created inside ‘Test Folder’ of Mickey, thus creating a hierarchy of folders:



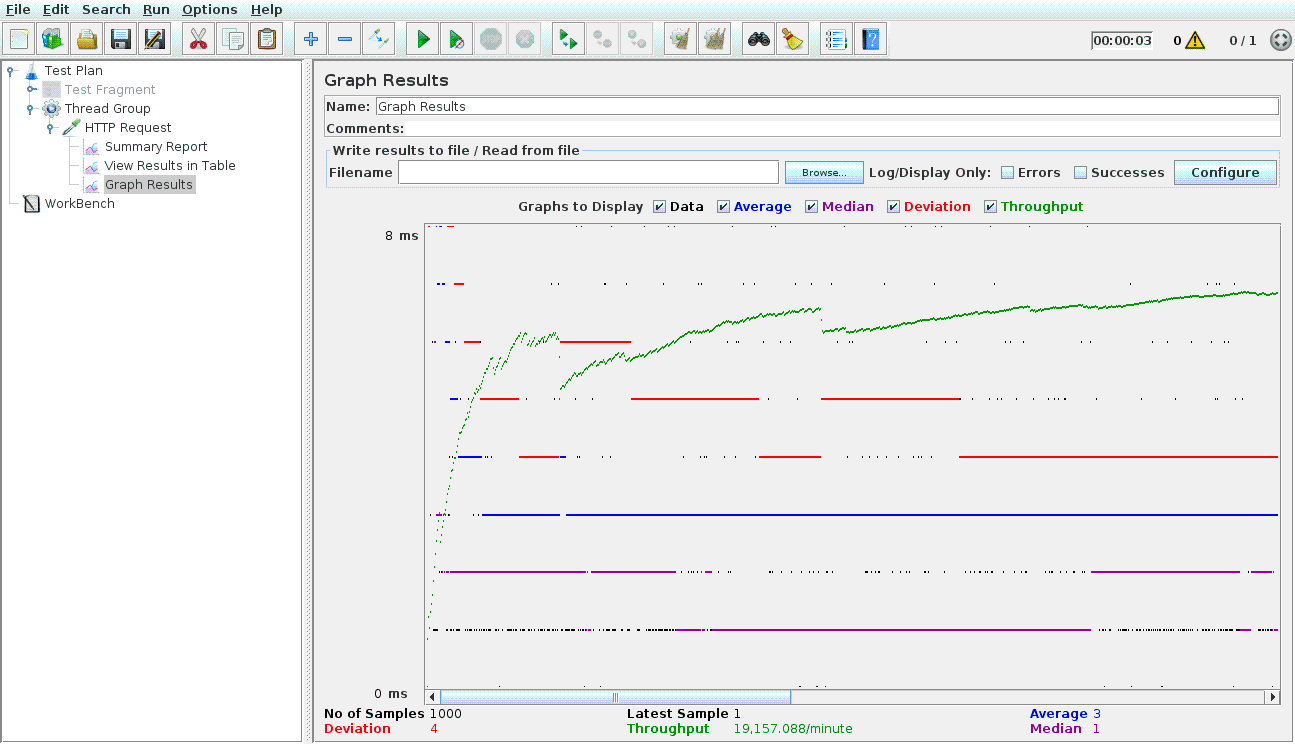
* Files can be uploaded within any folder:

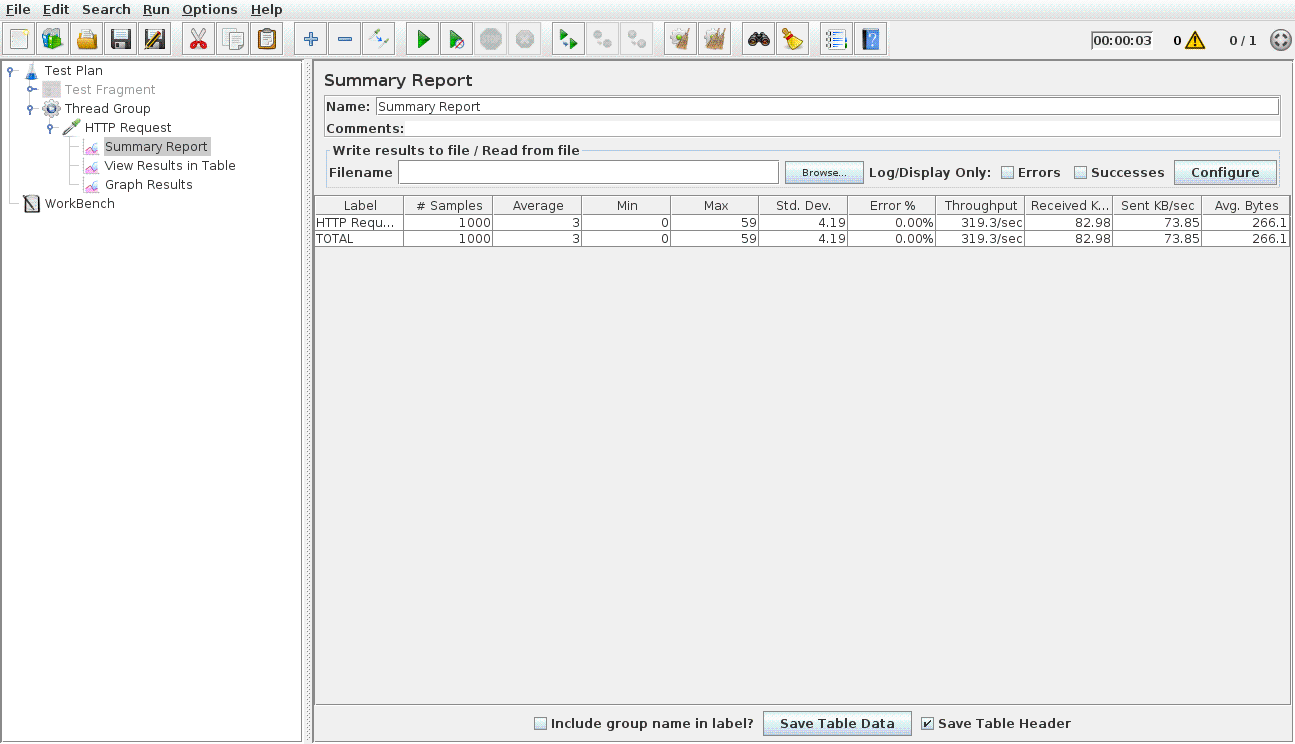


**Performance**

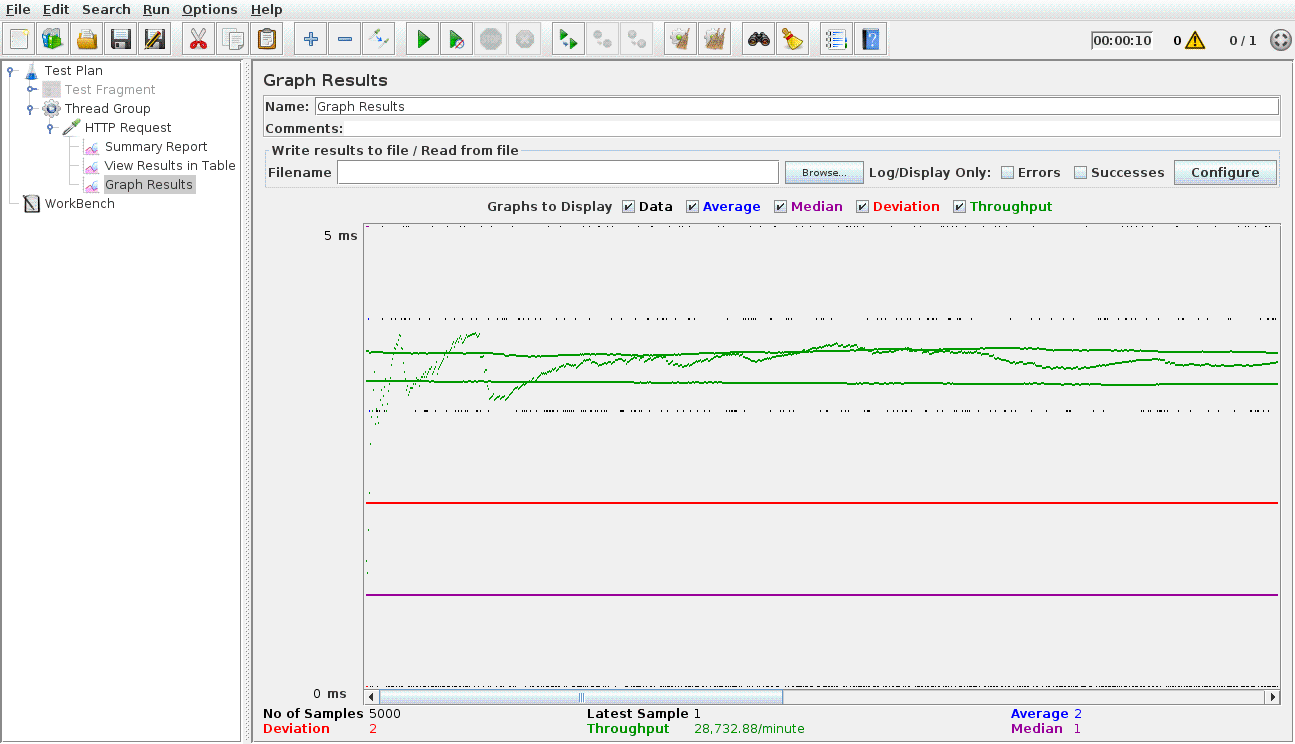
**Calculator:**

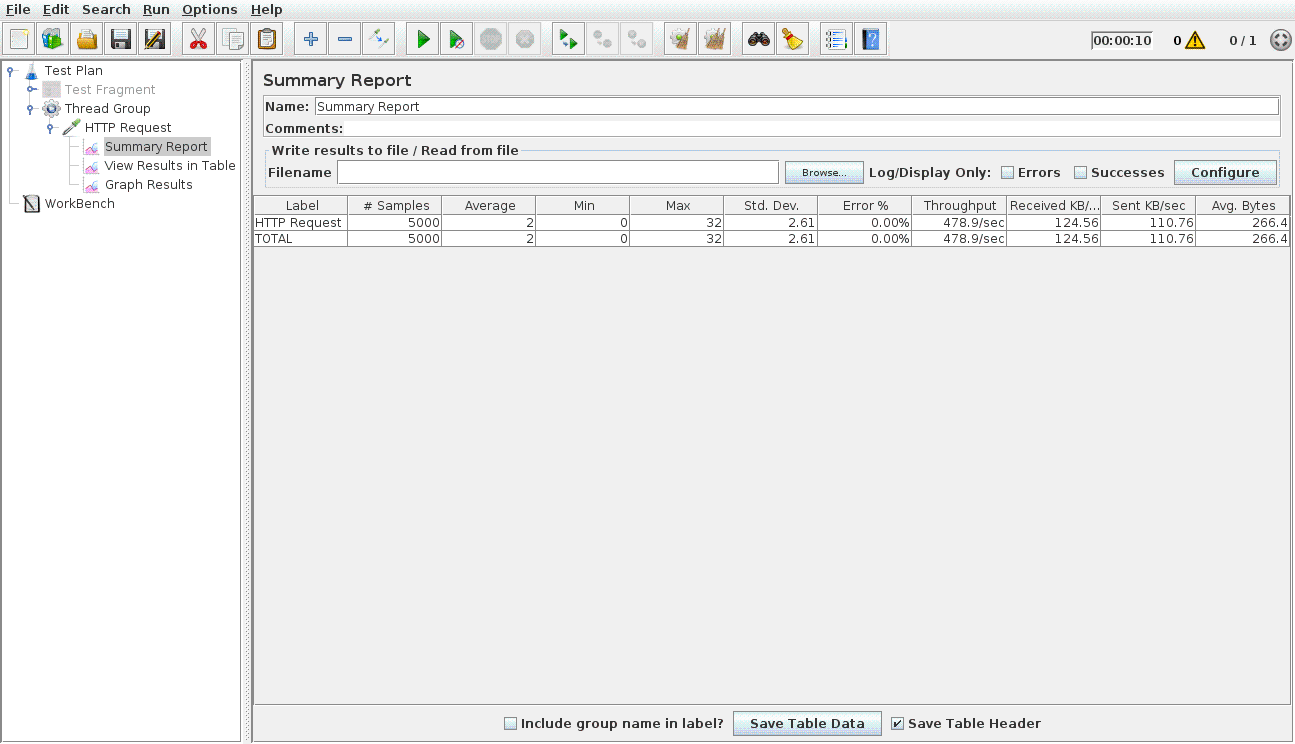
* 1000 calculator calls on randomly selected tasks:



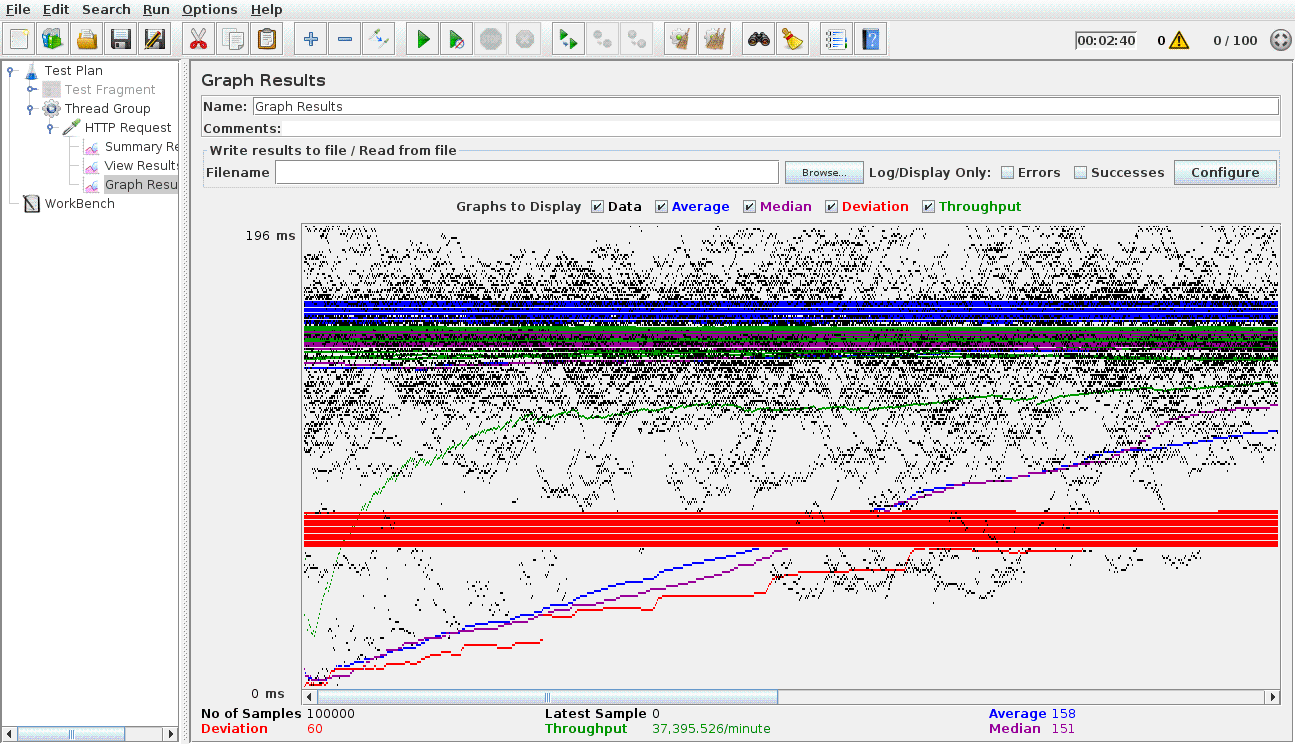


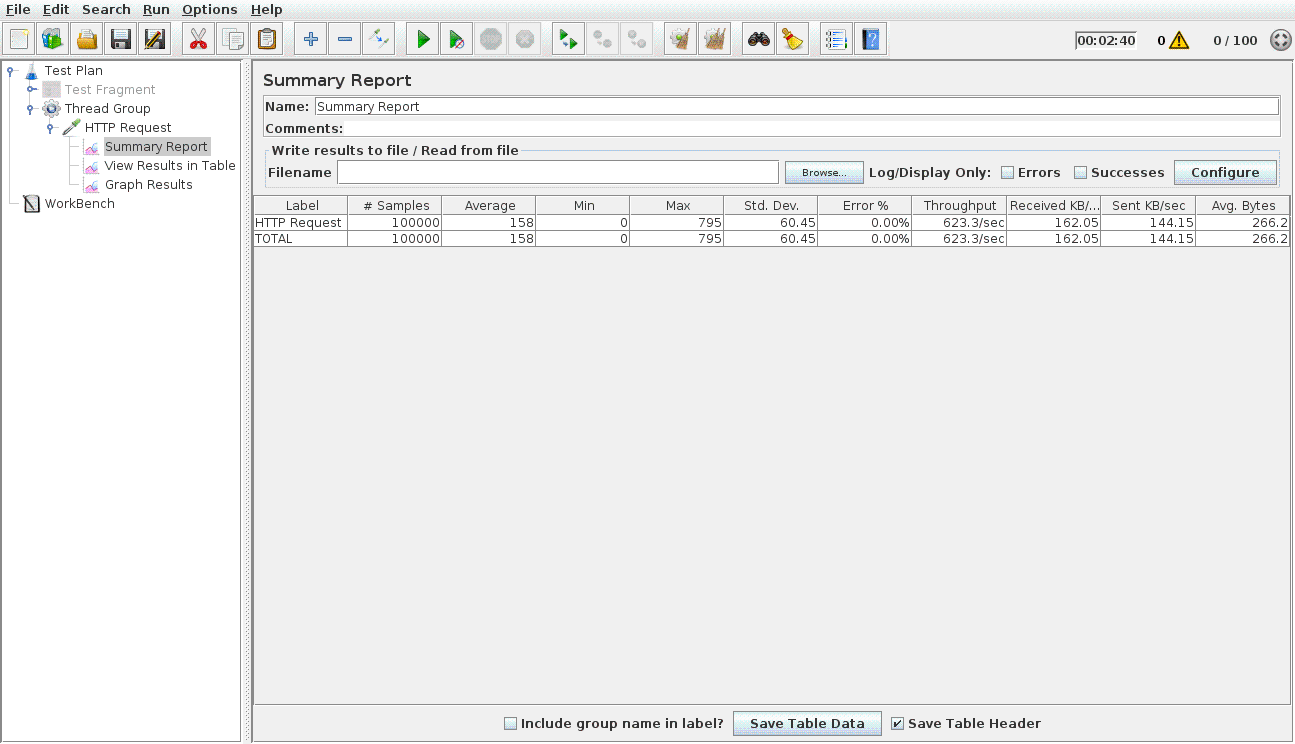
* 5000 calculator calls on randomly selected tasks





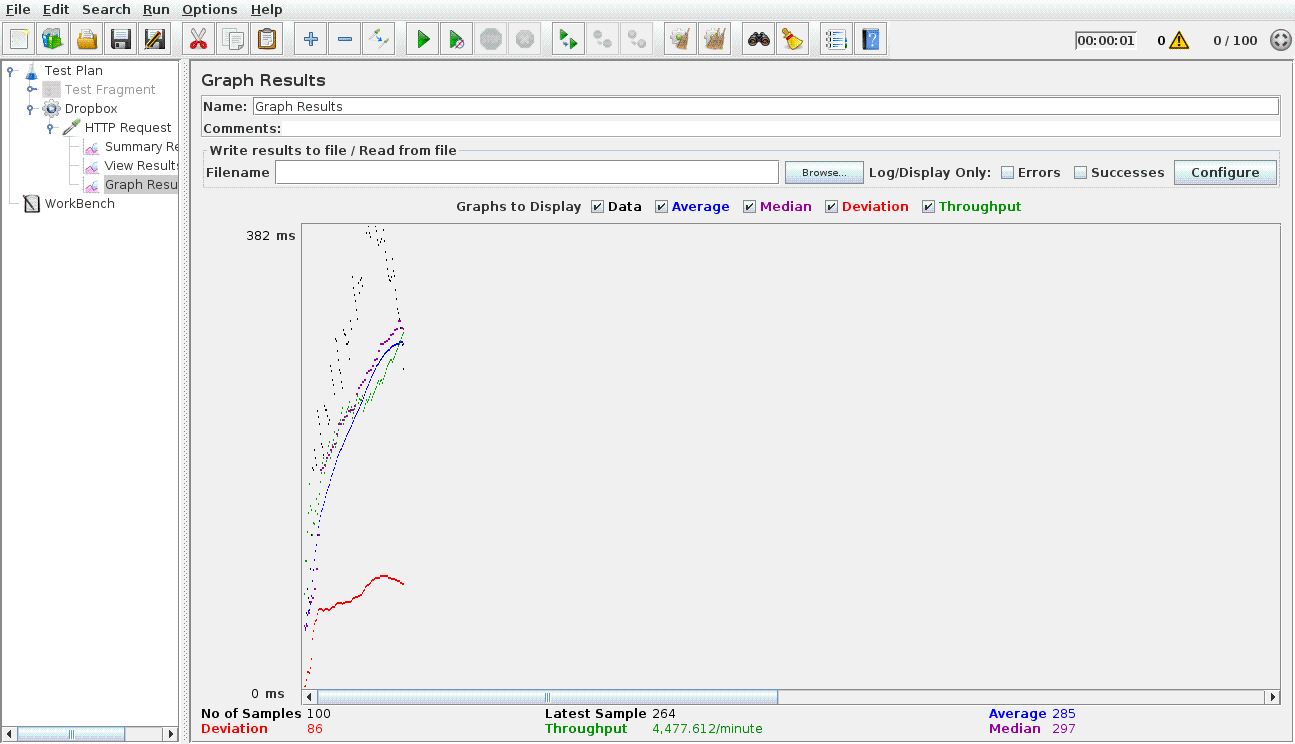
* 1000 calculator calls by 100 concurrent users:

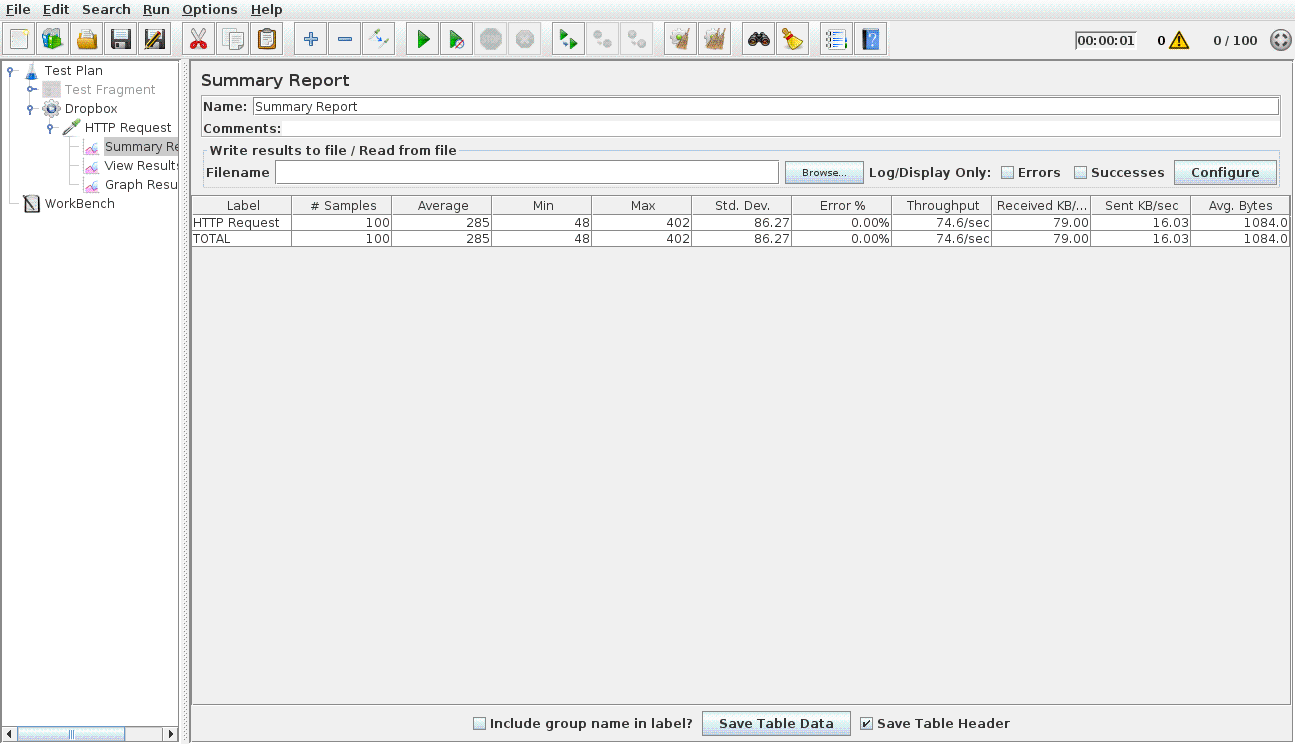




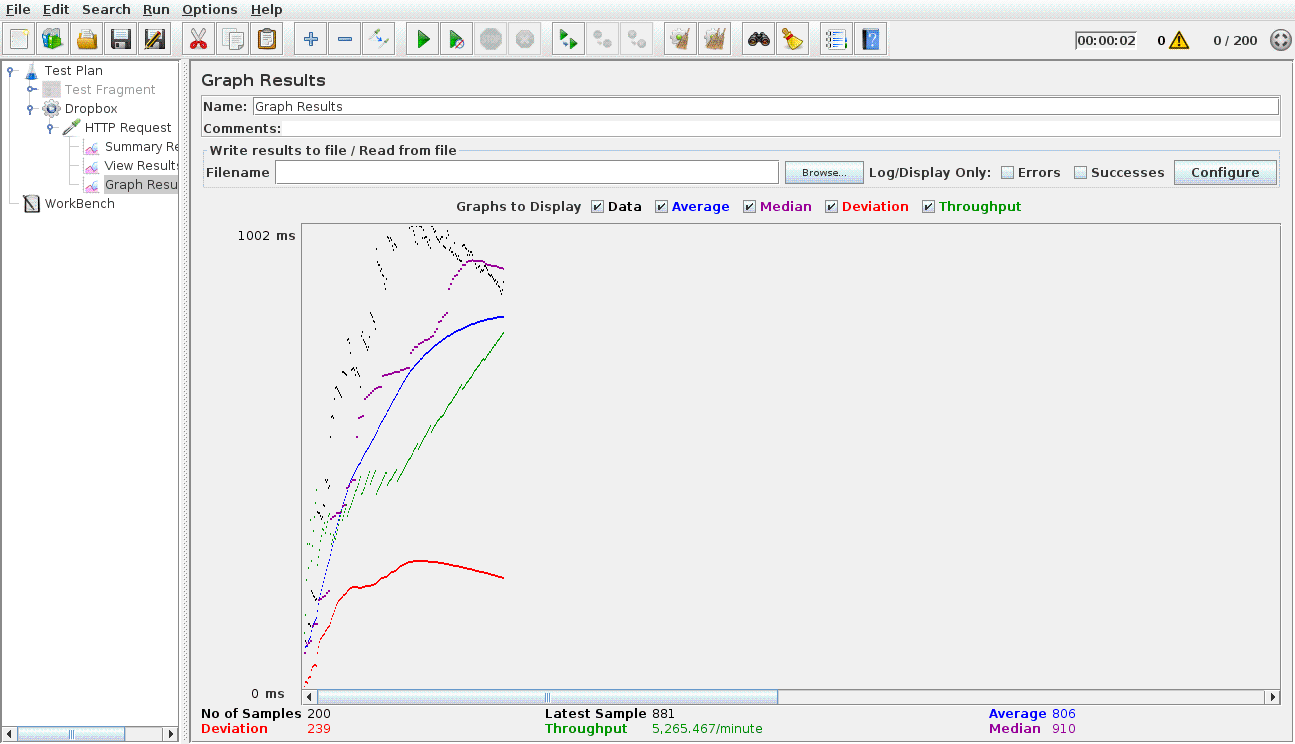
**Dropbox:**

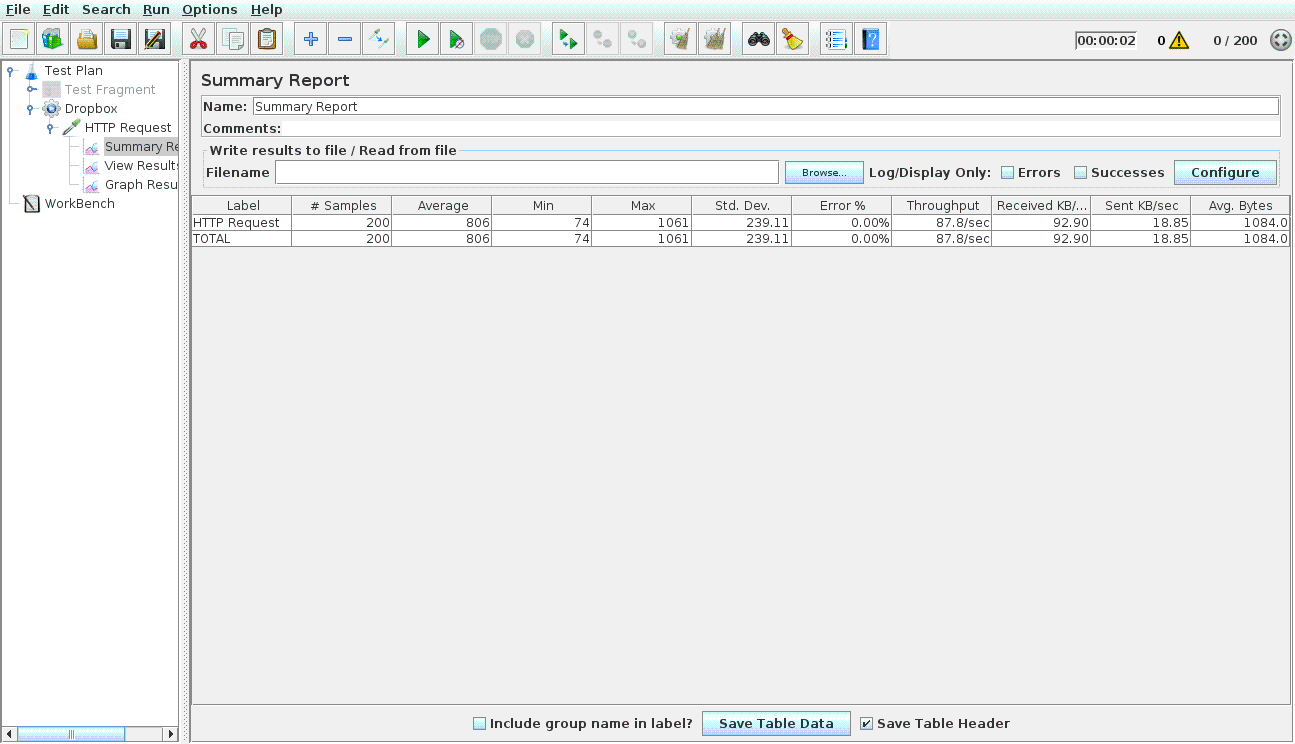
* **100 concurrent users**



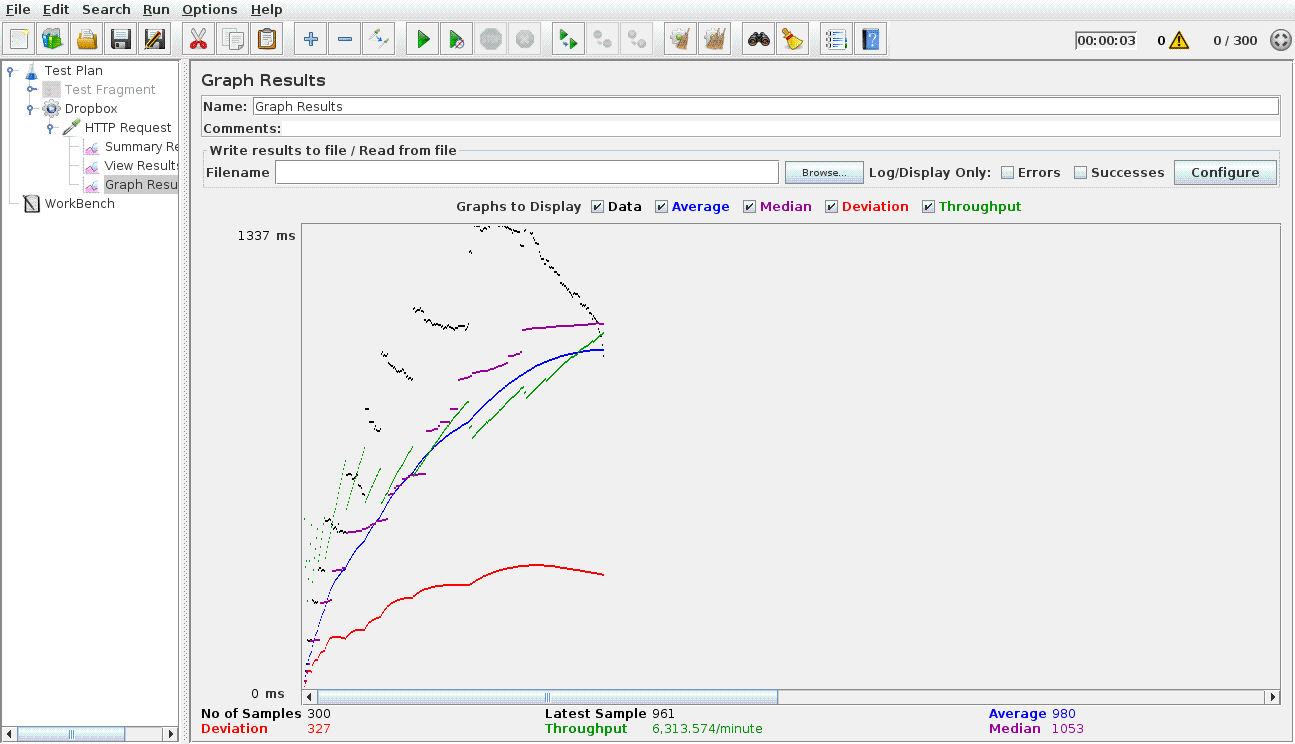


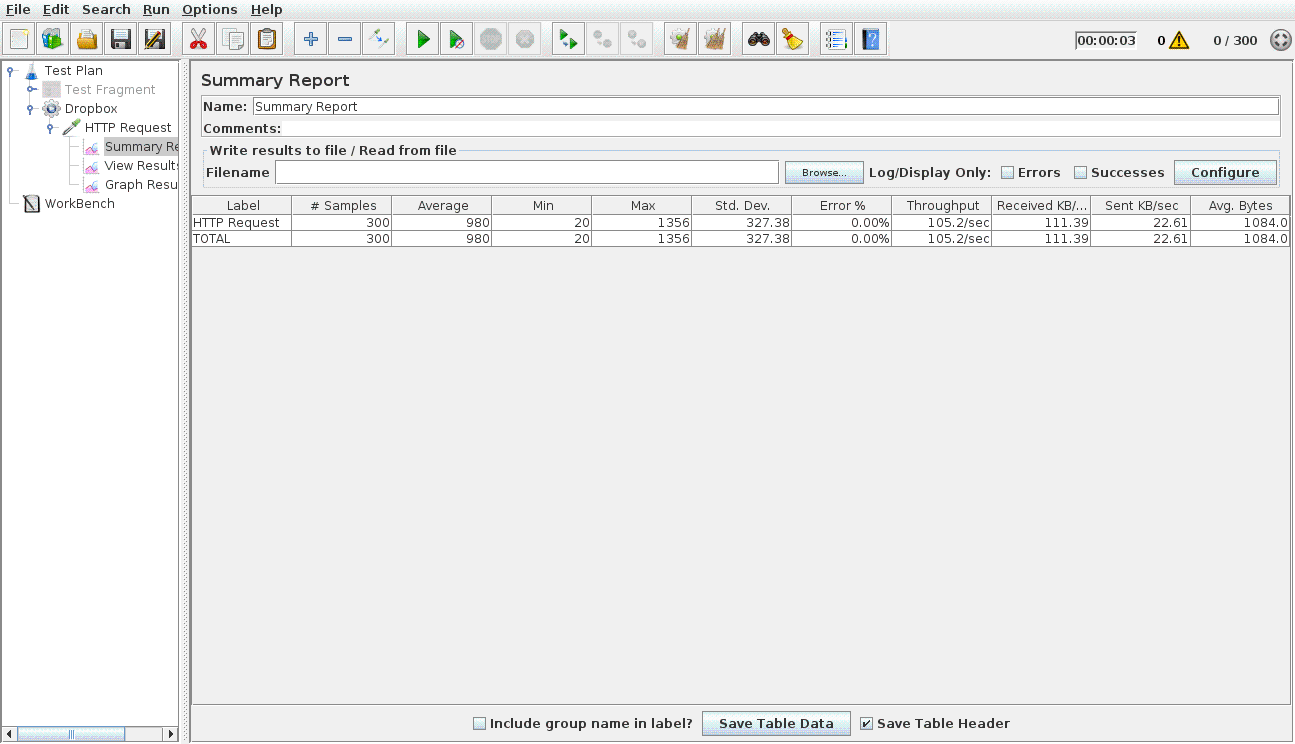
* **200 concurrent users**



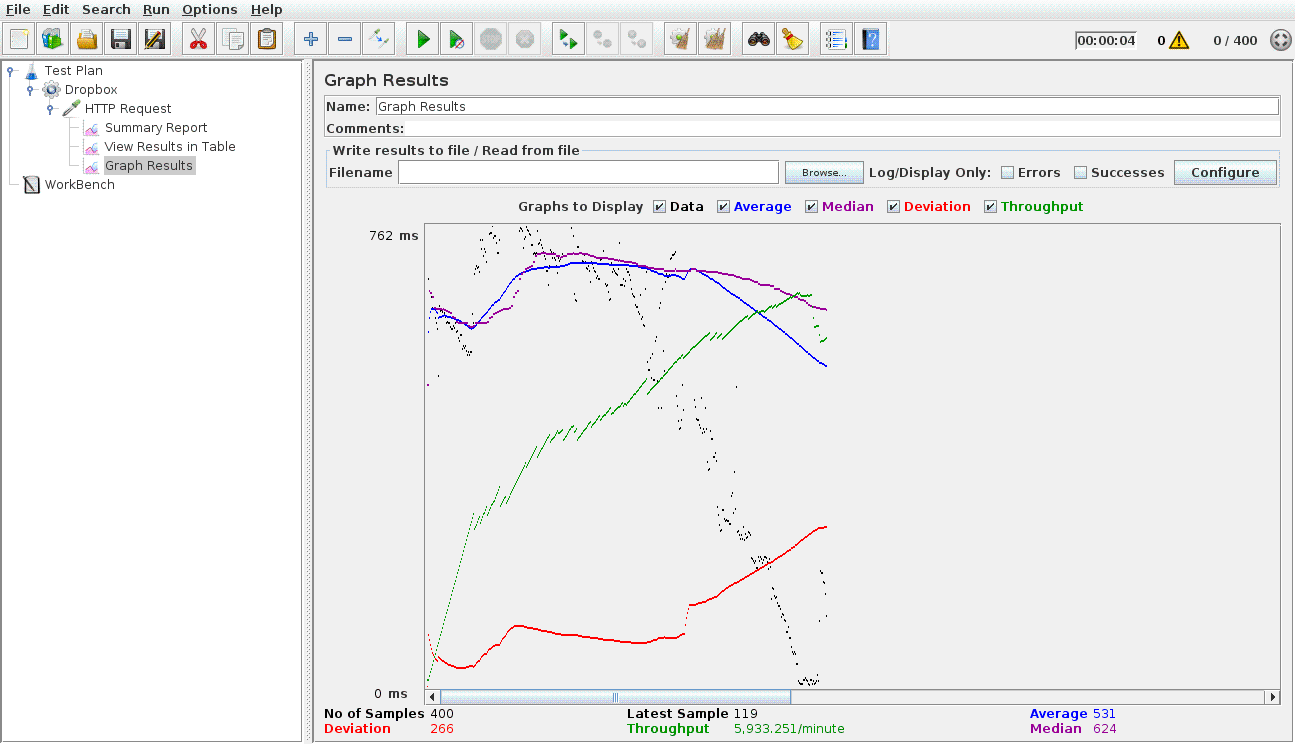


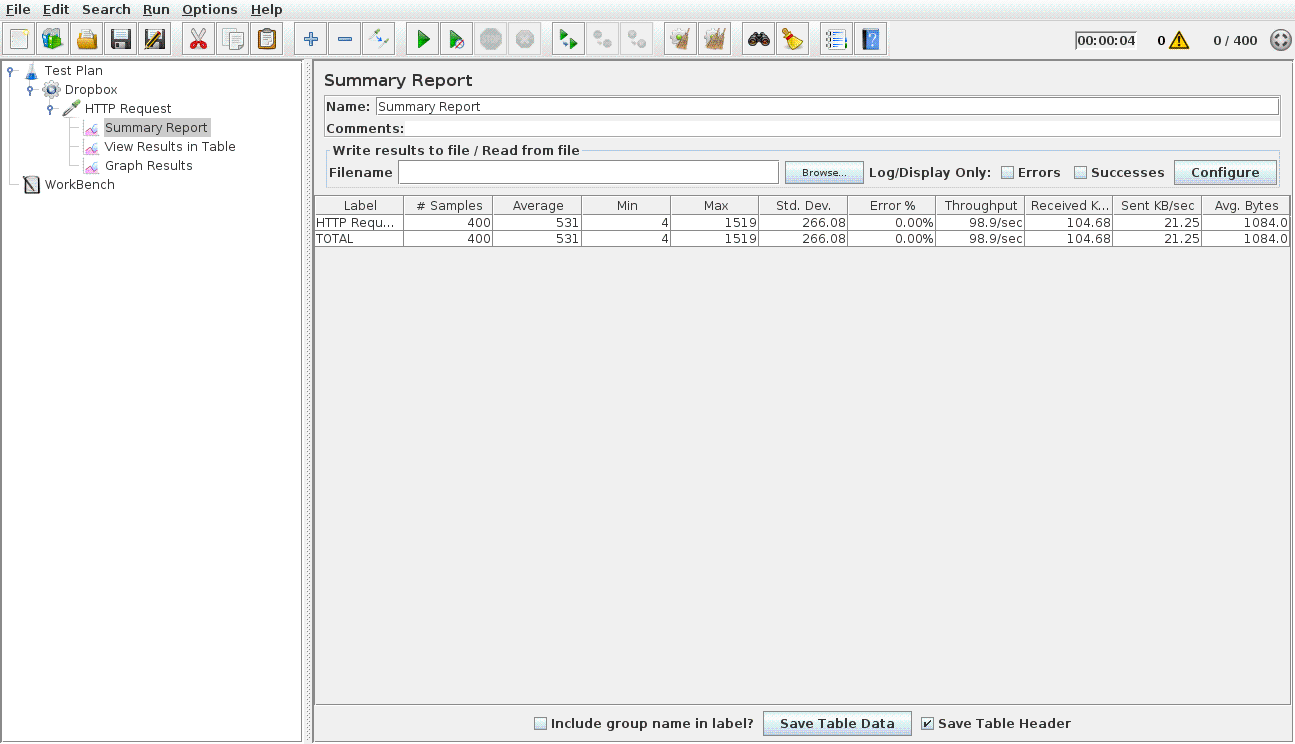
* **300 concurrent users**



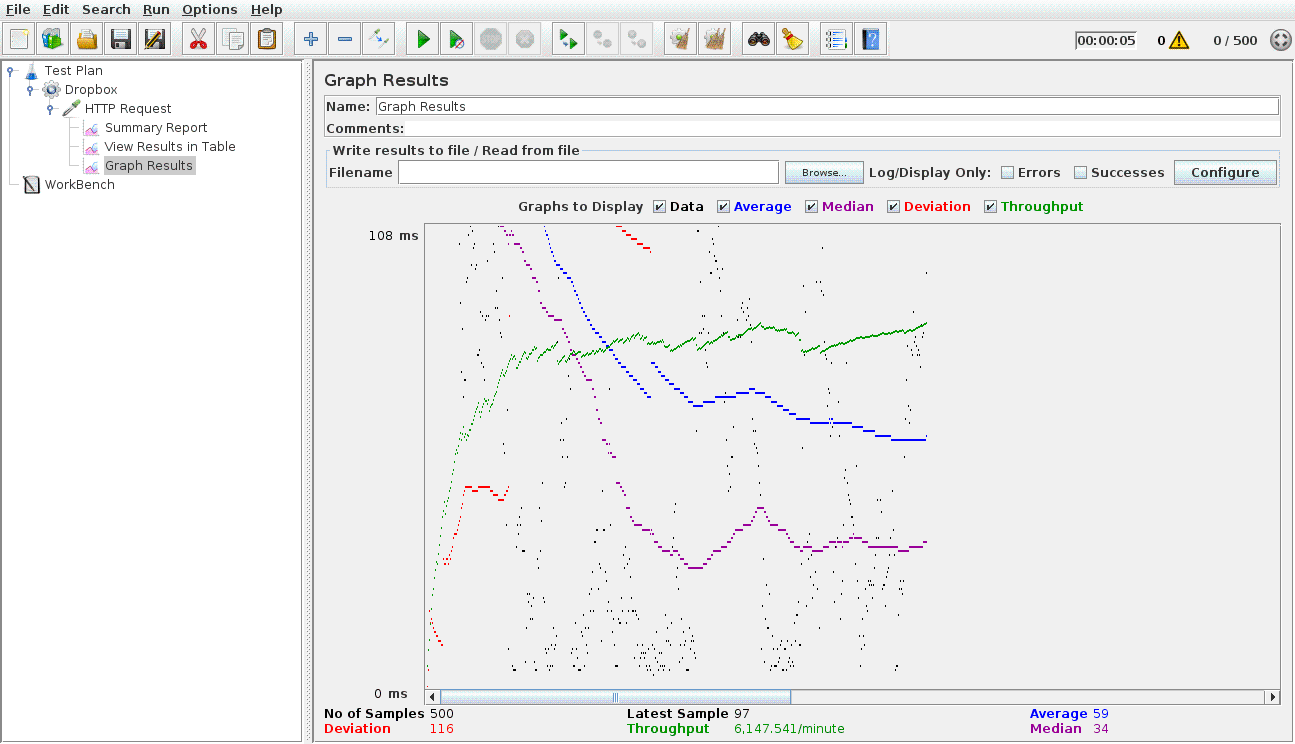


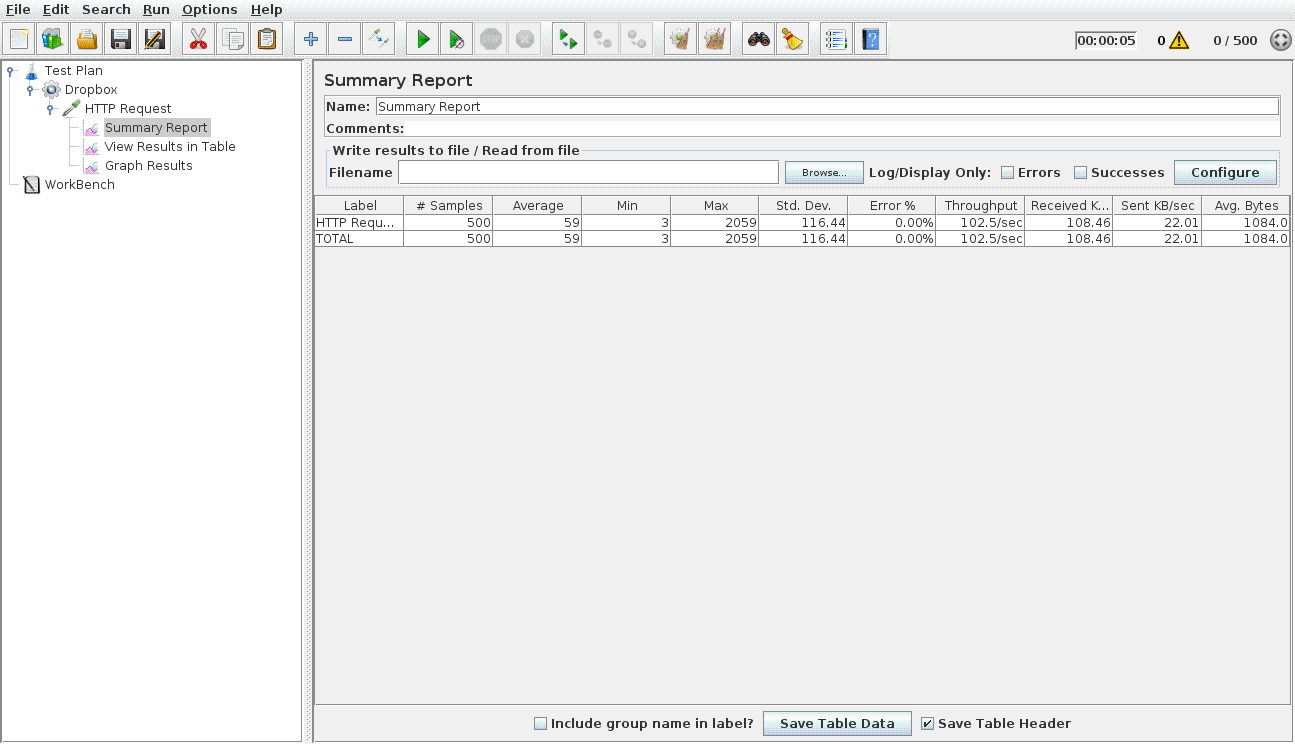
* **400 concurrent users**



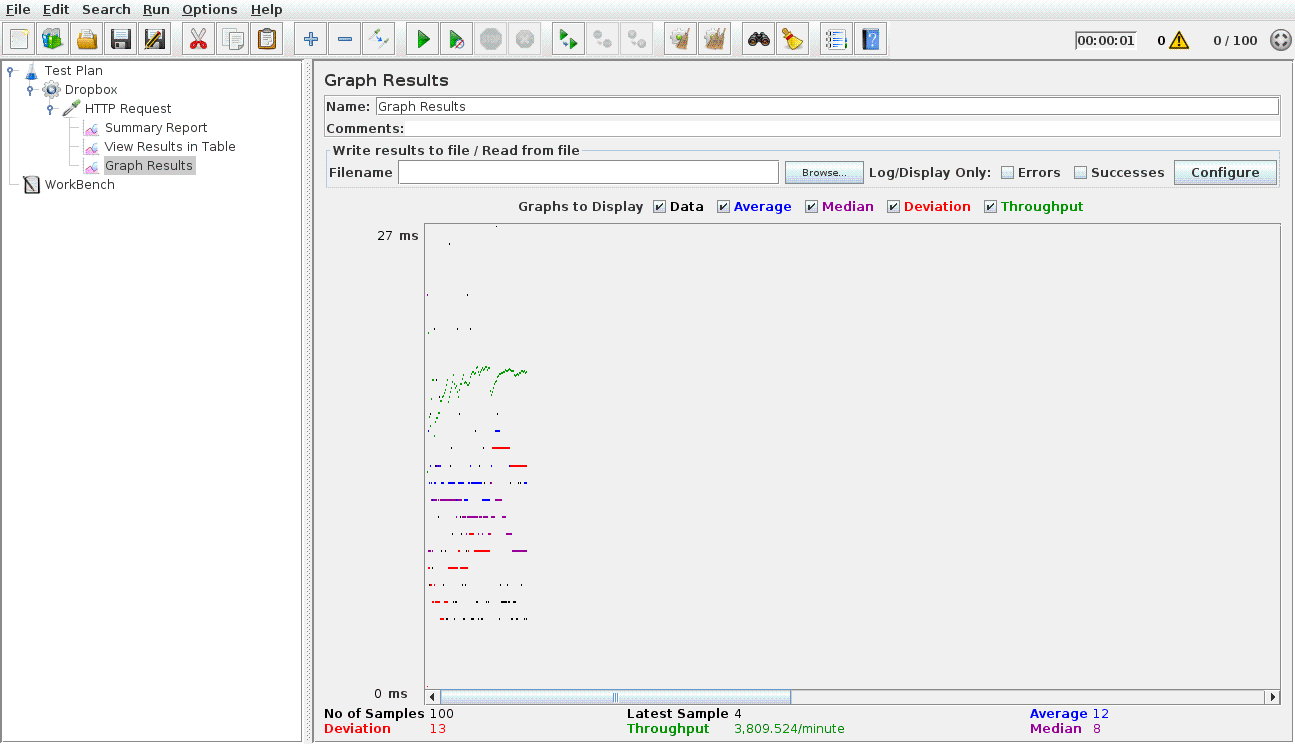


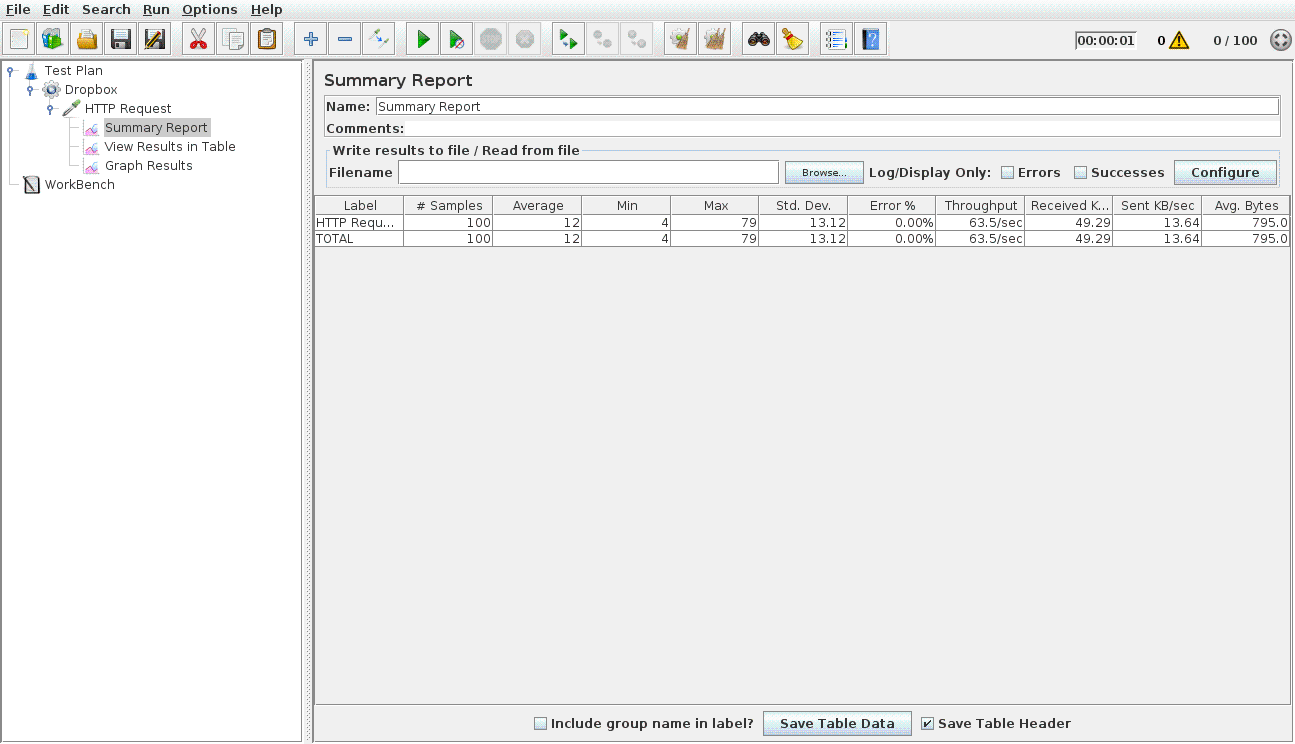
* **500 concurrent users**



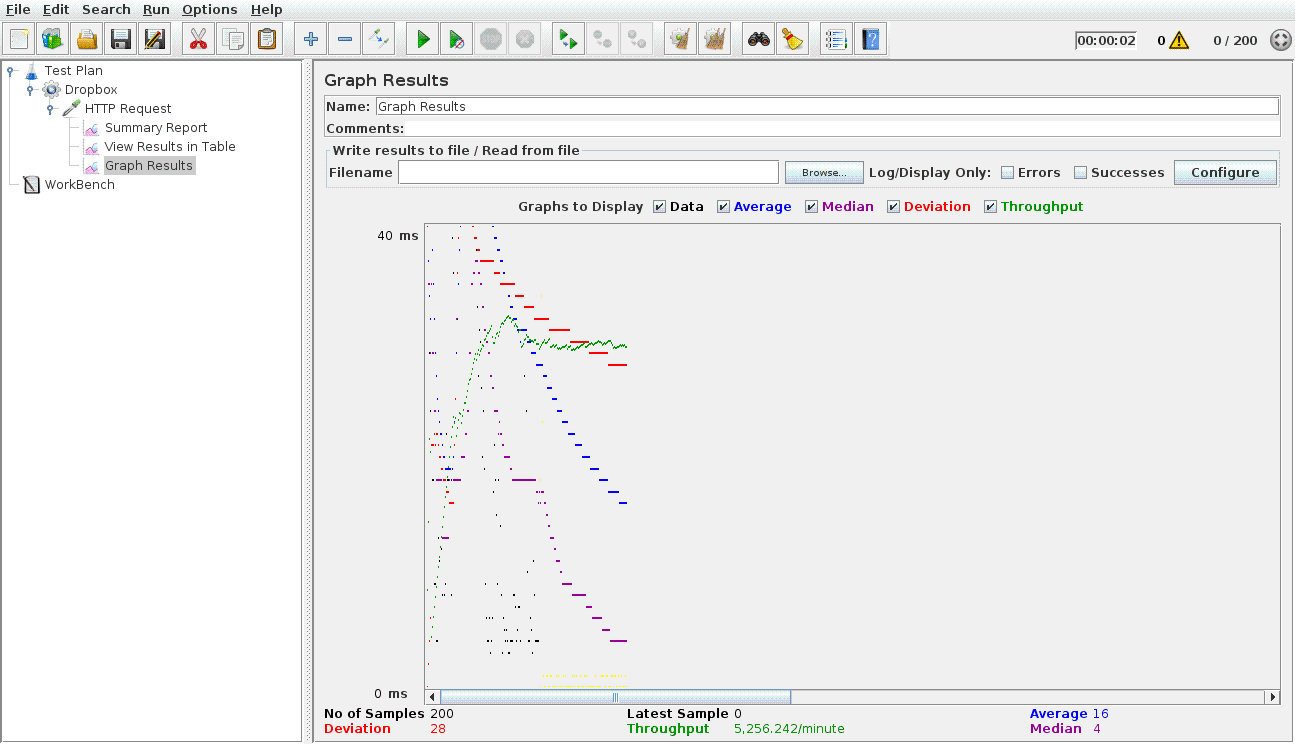


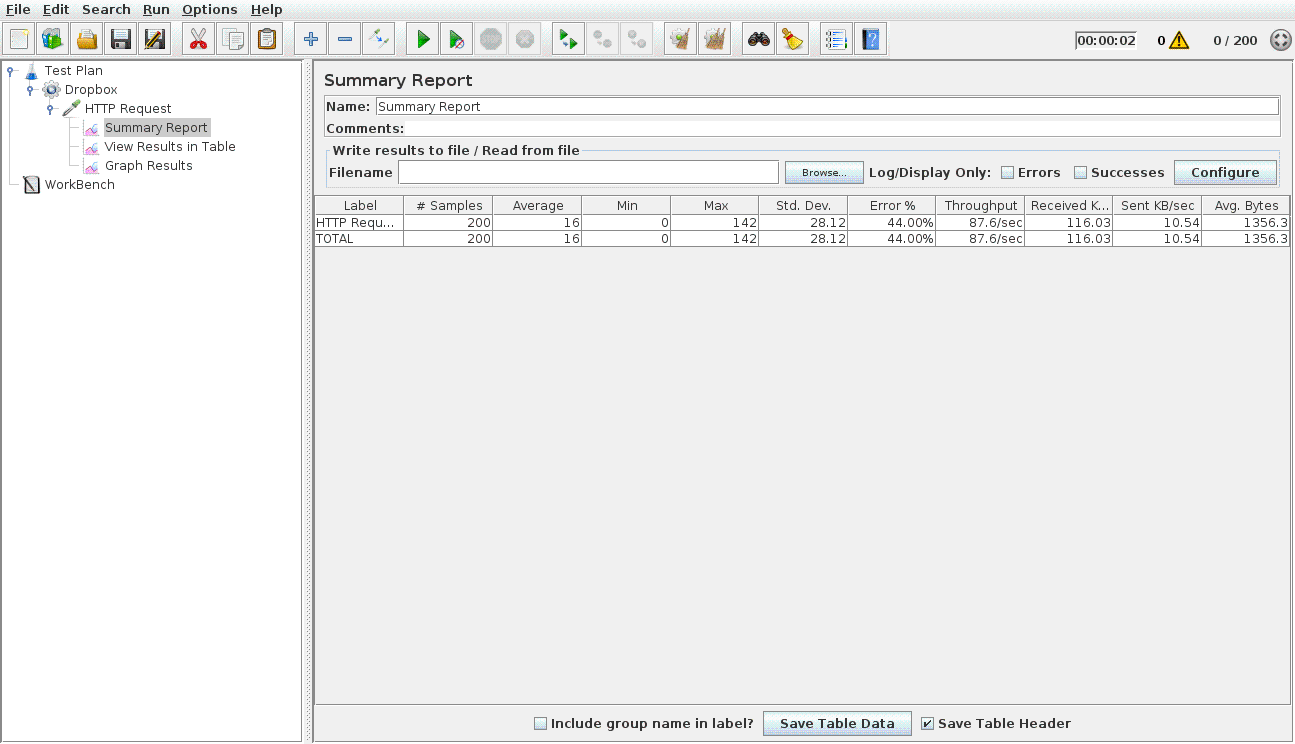
* **100 concurrent users with connection pooling**



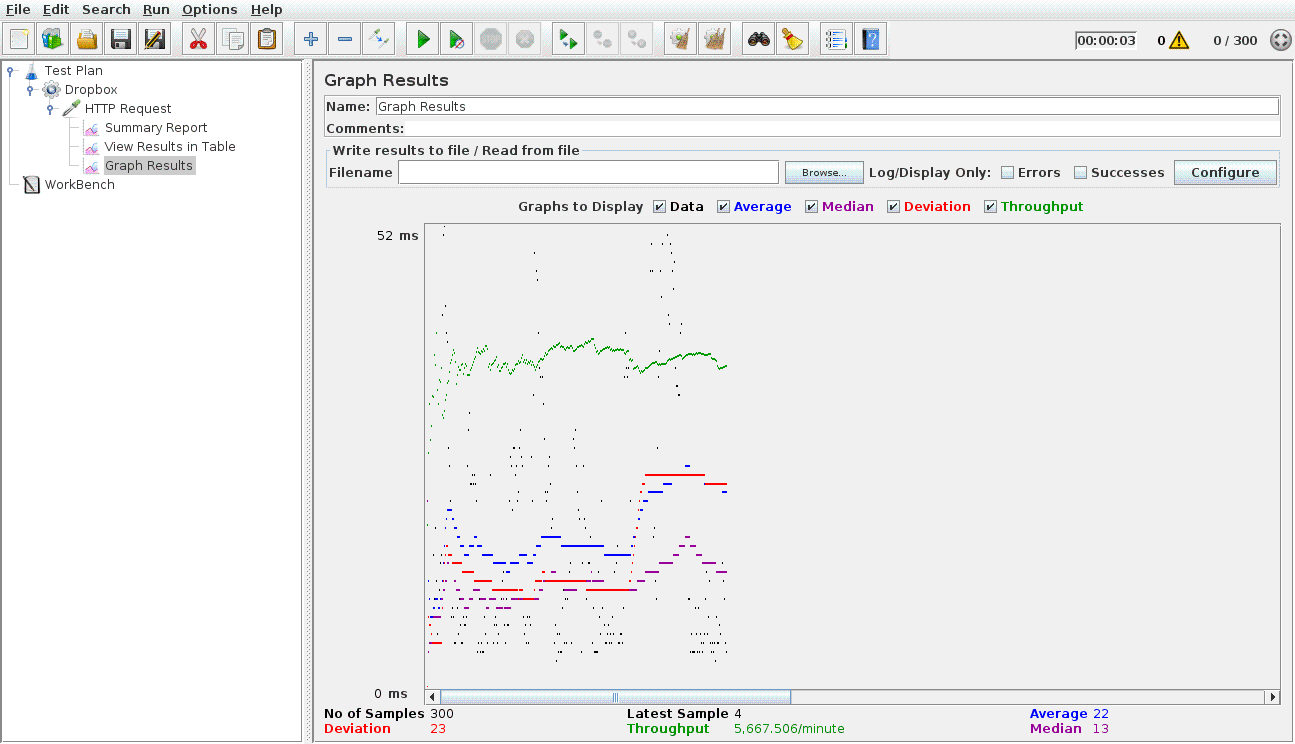


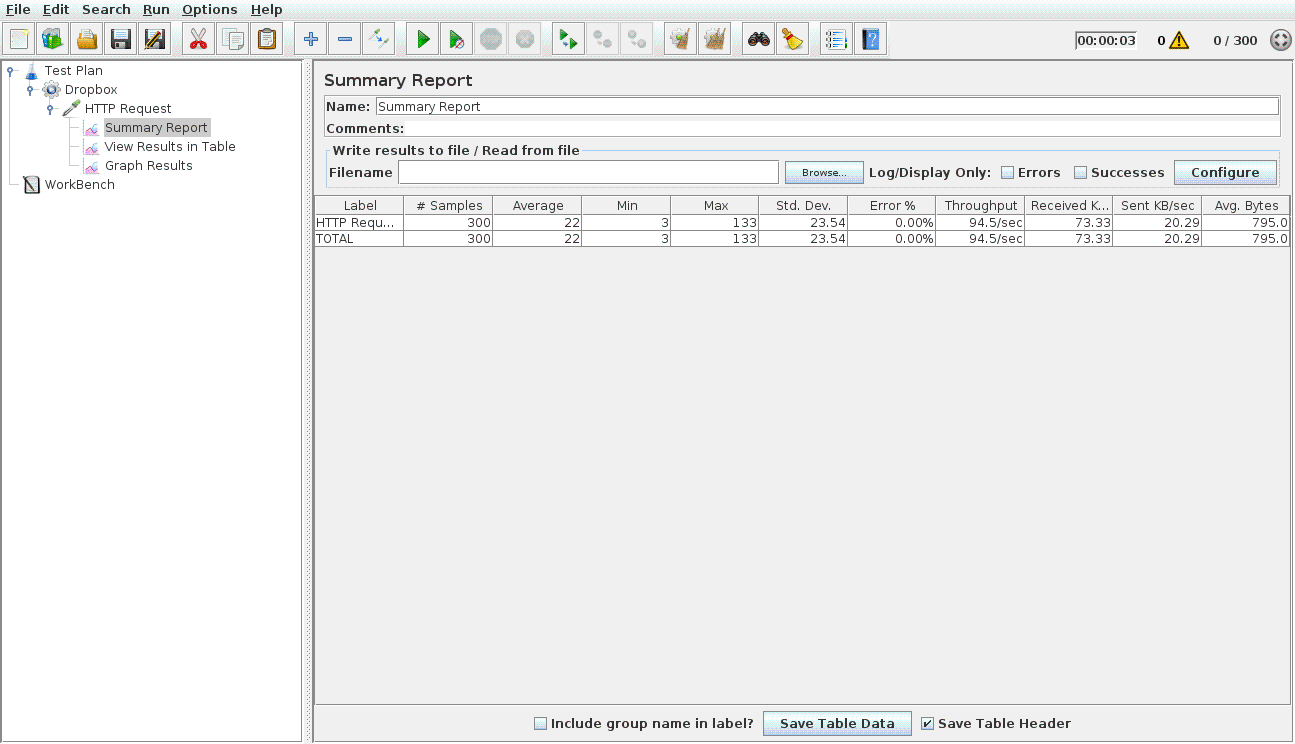
* **200 concurrent users with connection pooling**



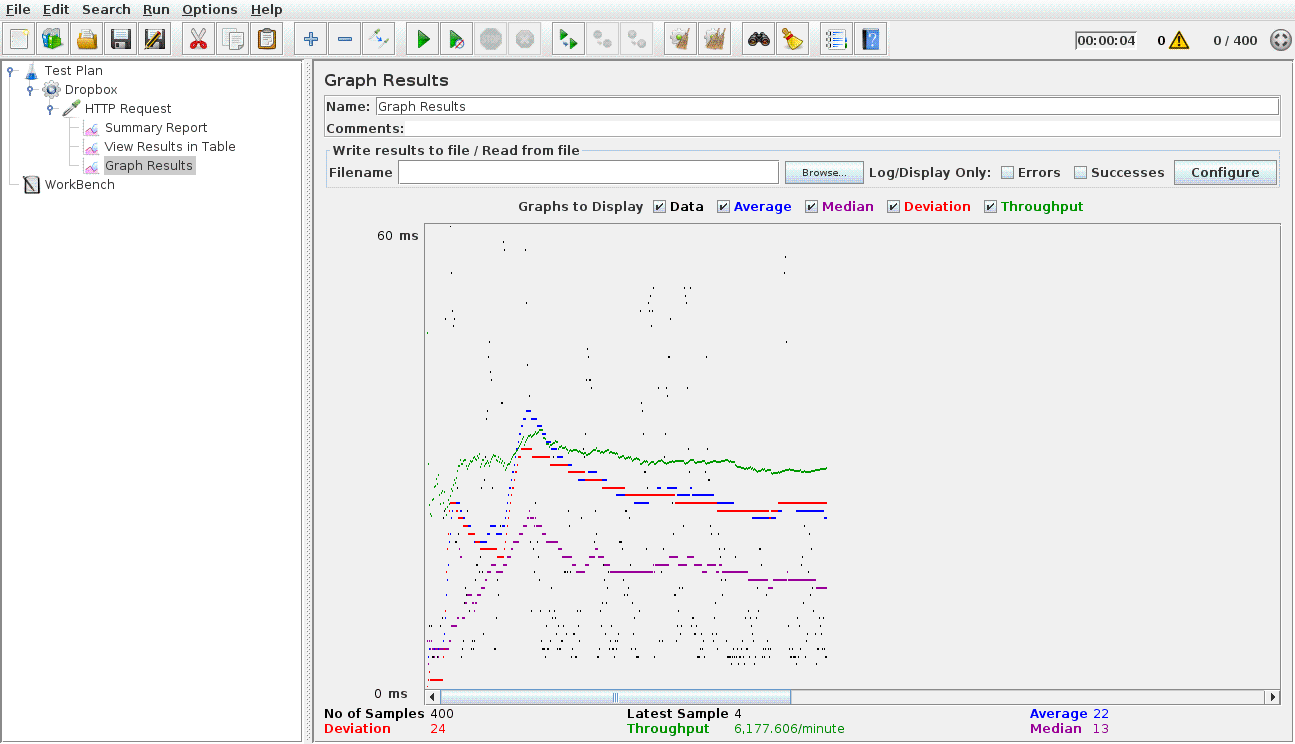


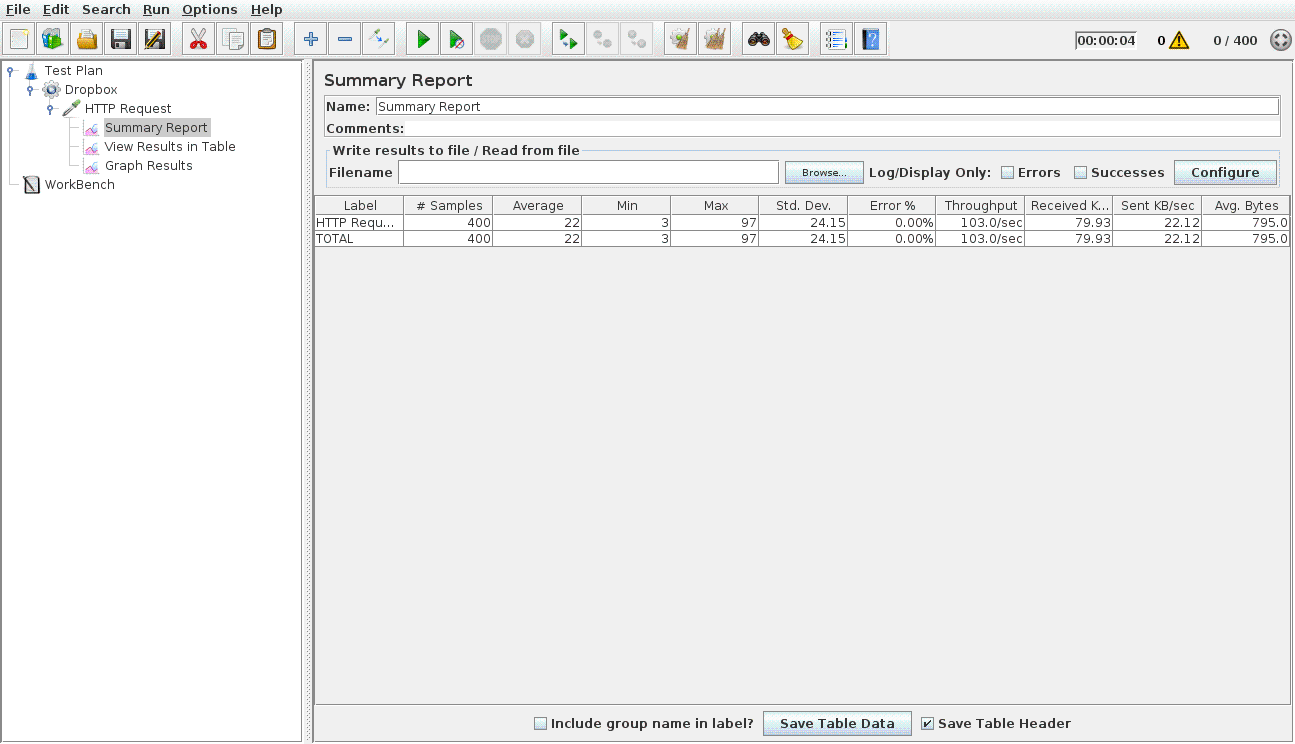
* **300 concurrent users with connection pooling**



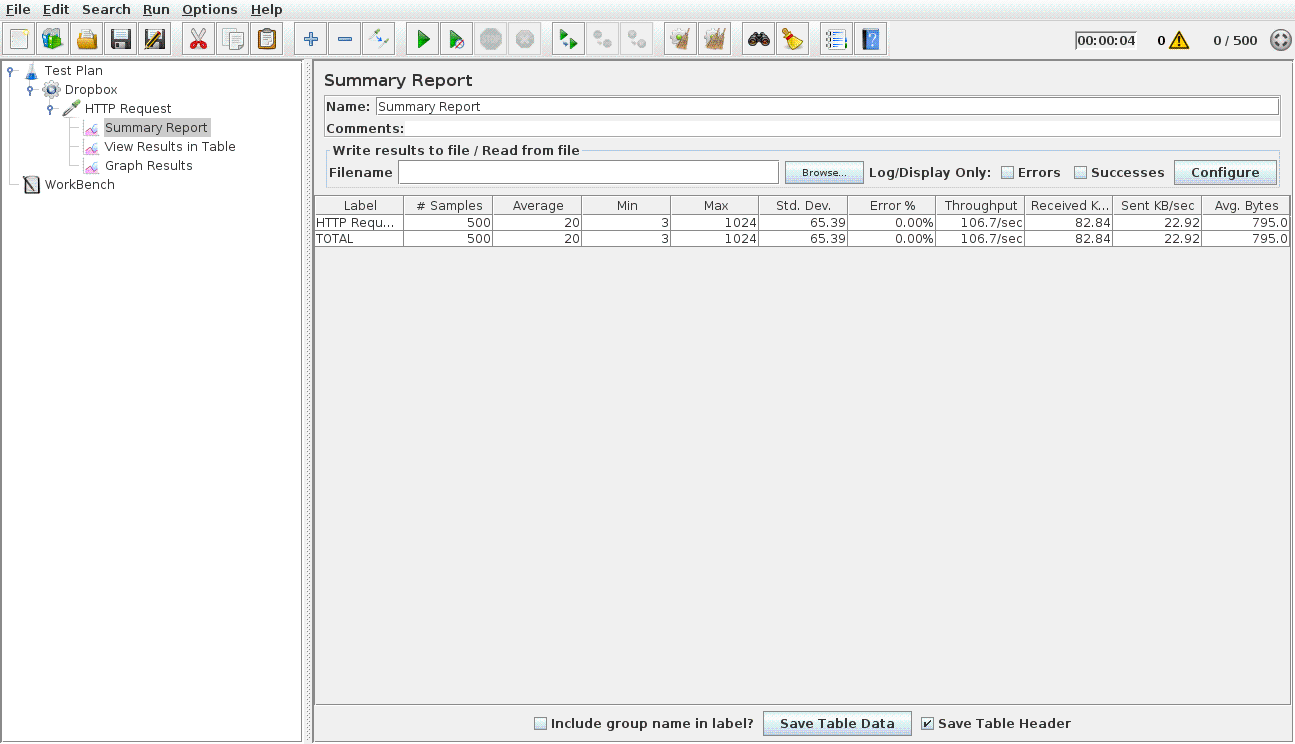
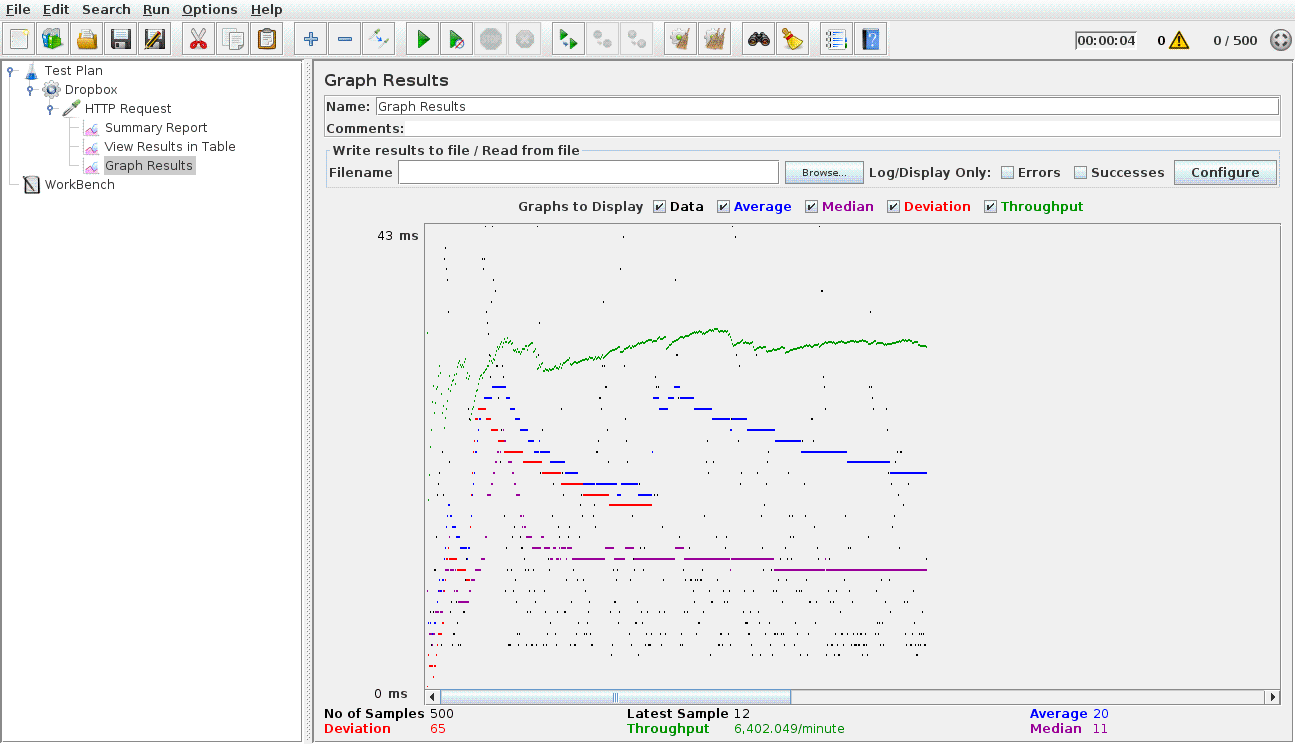


* **400 concurrent users with connection pooling**





* **500 concurrent users with connection pooling**



**Result Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Throughput (/minute) | Average (ms) | Median (ms) |
| 100 users | 4477 | 285 | 297 |
| 100 users with conn pooling | 3809 | 12 | 8 |
| 200 users | 5265 | 810 | 906 |
| 200 users with conn pooling | 5256 | 16 | 4 |
| 300 users | 6313 | 980 | 1056 |
| 300 users with conn pooling | 5667 | 22 | 13 |
| 400 users | 5933 | 531 | 624 |
| 400 users with conn pooling | 6177 | 22 | 13 |
| 500 users | 6147 | 59 | 34 |
| 500 users with conn pooling | 6402 | 20 | 11 |

* Here **throughput** is the number of requests per unit time that are sent to the server for testing. **Average** is the average response time of the of the total samples. **Median** is time taken for response to comeback from server. Median is the point where half the samples are above it and half the samples are below it.
* In the graphs above, throughput, average and median are shown by green, blue and purple lines respectively.
* The dropbox server is tested here for 100, 200, 300, 400, 500 concurrent users with and without connection pooling. All the users are making an API request to fetch the list of uploaded files during the test.
* As it can be seen from the graphs above, throughput remains fairly constant after the initial increase which shows a consistent system performance. At the beginning of the test, as the requests comes in, the throughput keeps increasing until the number of concurrent requests steadies.
* It is clearly observed from the summary table above that the use of connection pooling increases the performance of the server quite drastically. The average and median comes down from around 800-1000 ms to around 20 ms when connection pooling is implemented.
* Deviation is shown in Red in the graphs above. It indicated the variation from the average. It is observed from the graphs above that the deviation remains to be around 100 ms. Lower variation indicates consistancy.

**Questions**

**1.Explain the encryption algorithm used in your application. Mention different encryption**

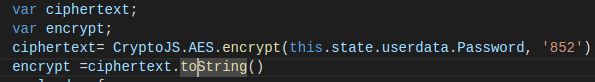
**algorithms available and the reason for your selection of the algorithm**

**used.**

**Ans.**

* As per the password encryption requirement, I have used the **AES (**Advanced Encrypted Standard**)** encription algorithm from **Cryto.js** libraries. Crypto.js is a collection of cryptographic algorithms implemented in Javascript.

Below is the screenshot from the login component of Client 2.



* **Encryption algorithms available** : MD5, AES, SHA-1, SHA-256, DES, MARC4, Rabbit, HMAC (HMAC-MD5, HMAC-SHA1, HMAC-SHA256) PBKDF2
* **Encryption libraries for JavaScript** : Stanford, ecmaScript, Forge, Crypto-js, Ohdave, pidCrypt, etc.
* **Reasons for choosing AES algorithm of Crypto-js :**
  + Crypto-js AES-128, AES-192 and AES-256 and it automatically picks up the variant according to the key passed in which in turn makes it difficult to brute the key.
  + AES is faster in both software and hardware and provides consistent and simple interface.
  + Encrypting and decrypting AES key data is relatively easy as compared to other public key algorithms.
  + Even with the computers running at the 17 petaflops is essentially unbreakable.
  + **Such security and simple interface provided by the AES makes it perfect to be used for the Password encryption in this lab.**

**2.Compare the results of graphs with and without connection pooling of database. Explain**

**the result in detail and describe the connection pooling algorithm used in your code.**

**Ans.**

* **Comparison** of results with and without connection pooling for Dropbox.

|  |  |  |
| --- | --- | --- |
| No. of Users | Average Time (ms) | |
| Without Connection Pooling | With Connection Pooling |
| 100 | 285 | 12 |
| 200 | 806 | 16 |
| 300 | 980 | 22 |
| 400 | 531 | 22 |
| 500 | 590 | 20 |

* As it is clearly visible from the above table, using connection pooling increases the performance of the database.
* When connection pooling is not used, each connection is created when there is a need for one and the connection is closed once it has served the request. This brings the overhead of creating a new connection at the time of executing a query. This slows down the database and thus the degrade in performance.
* While in connection pooling, a group of connections are created while creating the pool and are re-used. Connection pool pre-creates the connection and when one of them is requested, it is used from the pool where it has already gone through process of creation. This leads to lesser required time for every request. Once the request is completed the connection is simply released and can be used by another request later.
* **Connection Pooling algorithm**
  + Create connection using mysql.createPool.
  + Configure the pool as per requirements.
  + Everytime a request comes, getConnection() from the pool
  + After serving the request,release the connection using connection.release();

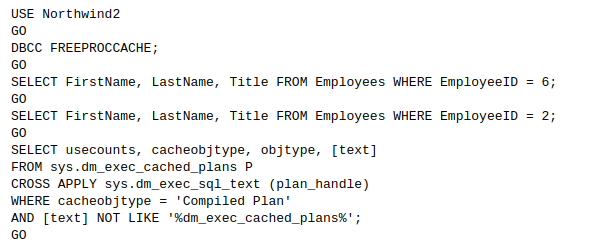
**3.What is SQL caching? What all types of SQL caching is available and which suits your code**

**the most. You don’t need to implement the caching, write pseudo code or explain in**

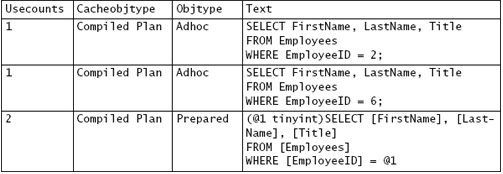
**detail.**

**Ans.**

* **SQL caching :** Complications of the previously executed queries can be avoided by using the SQL caching. SQL Server cache the statements/queries executed overtime in a store. Each time a statement is executed, the server first looks in to the cached sql statements plans, if the plan is already cached, server uses that already compiled plan for the execution rather than compiling a new plan for each execution. This mechnism of plan caching to save the complications of already executed queries is called SQL Caching.
* Benefits :
  + Imporves data access performance
  + Reduces database load
  + Cache servers distribute the load of repetitive queries.
  + Reduces the number of queries hitting the database.
* **Types of SQL Caching :**
  + Adhoc query caching
  + Autoparameterization
  + Prepared queries
  + Stored procedures
* From the above types of Caching, **Autoparameterization** seems to be the most suitable for my code. Autoparameterization uses the same query plan for similarly structured queries with different parameters.
  + In our case, many query structures, such as, updating the files and folders in the database, also fetching a user’s files and files shared with the users, all these have similar structures and can be cached using the Autoparameterization.
* Autoparameterization can be implemented as mentioned in below sample:



* + Executing above code returns three rows as shown below.



* + Individual queries with distinct constants do get cached as adhoc queries.
  + The third row returned from sys.dm\_exec\_cached\_plans has an objtype of prepared. The query plan is associated with the prepared plan, and it can be observed that the plan was used twice.
  + Autoparameterization is also disallowed for data modification statements that use DELETE statements with FROM clause.

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

**Ans.**

* In my Dropbox application for lab-1, I have implemented the session management strategy using localstorage of the browser. For a session to be horizontally scalable, it should be able to persists even when it is requesting on different servers.
* Session handling strategy used here for lab-1 involves storing the user in the local storage of the browser being used. Thus, this is the client side session management. The session management implement is explained below.
  + When the user first logs in, he is saved in the localstorage to maintain the session. All the subsequent calls to server are implemented for this session.
  + Even if the client closes the browser tab, he will be directly be logged into the application as the session is still maintained. The session is terminated once the user logs out of the application and the localstorage of the user is removed.
  + Whenever a new request arrives on the login page, it checks in the localstorage whether the session is maintained or not. If yes, user is redirected to the homepage or else the user has to log in to create a new session.
  + The strategy used for sessions is horizontally sclable as the session is being maintained from the client side only and can maintain the session even if the requests are served by multiple servers.

**References:**

https://msdn.microsoft.com/en-us/library/cc293623.aspx