# CMPE273: Enterprise Distributed Systems Lab 2 Kafka And MongoDB

****

**By: Sunny Udhani**

**Student ID: 012457289**

**Repository: https://github.com/sunny-udhani/online-file-sharing**

## **Introduction:**

Designed and developed a prototype of dropbox to demonstrate RESTful web services with Apache Kafka for distributed message streaming, MongoDB as datastore and PassportJS as authentication middleware.

My application includes 20+ kafka topics responsible to perform different activities. There are different consumers for each topic so that no one consumer gets overloaded with large number of requests on multiple topics. The system architecture is divided into 3 parts,

1. dropbox\_react
2. dropbox-kafka-client
3. dropbox-kafka-server

Here, the kafka-client part only does the part of producing messages and on response from kafka-server forward it to react side.

It also includes implementation of passportjs’s local strategy for authentication. Sessions are stored in MongoDB and I have used express-sessions for session management.

*Goals:*

* + - * Learn distributed service application development.
      * Understand the uses and applications of Non-relational databases.
      * Incorporate Enterprise application development techniques.
      * Design and implement a distributed service oriented application with modularized project files and functionalities.
      * Implement software design principles of high cohesion and low coupling.

*Purpose of System:*

* The system tries to imitate the services provided by dropbox.
* It allows the users to upload and share files, monitors user activity, etc.

The application provides features like,

1. Basic User-related functionalities
   1. Registration
   2. Login
   3. Logout
2. User Profile and Activity
3. File Upload (single & multiple)
4. Create Folder
5. Star – Unstar files/folder
6. Share file/folder by email/username.
7. List user-related files
8. List user groups
9. Create a group
10. Add/Delete members from group
11. Upload files in group (group share)(single & multiple files)
12. List files shared in group

## **System Design:**

Applications uses a simple Client-Server architecture where there are as many as 13 React components, 17 API’s and 20+ kafka topics to support different functionalities.

**Technology Stack:**

* Database used: MongoDB
* Front-End: HTML, Bootstrap and ReactJS
* Server-side: NodeJS and ExpressJS
* Message Streaming: Apache Kafka
* Authentication Middleware: PassportJS
* Session Management: express-sessions and MongoDB
* Package used for encrypting password: bcrypt (<https://www.npmjs.com/package/bcrypt>)
* Testing: MochaJS and Apache JMeter.

### Kafka Client – Server Interaction

kafka-consumer

Kafka Topics

kafka-consumer

kafka-consumer

kafka-consumer

register\_request

register\_resest

kafka-consumer

kafka-consumer

login\_request

kafka-consumer

kafka-consumer

login\_respon

kafka-consumer

kafka-consumer

kafka-producer

uploadFiles\_req

kafka-producer

DROPBOX-REACT

DROPBOX-KAFKA-CLIENT

DROPBOX-KAFKA-SERVER

uploadFiles\_res

kafka-consumer

kafka-consumer

makeDirectory\_request

kafka-consumer

kafka-consumer

makeDirectory\_res

kafka-consumer

kafka-consumer

starFile\_request

kafka-consumer

kafka-consumer

kafka-consumer

starFile\_respons

kafka-consumer

kafka-consumer

listGroups\_request

kafka-consumer

listGroups\_response

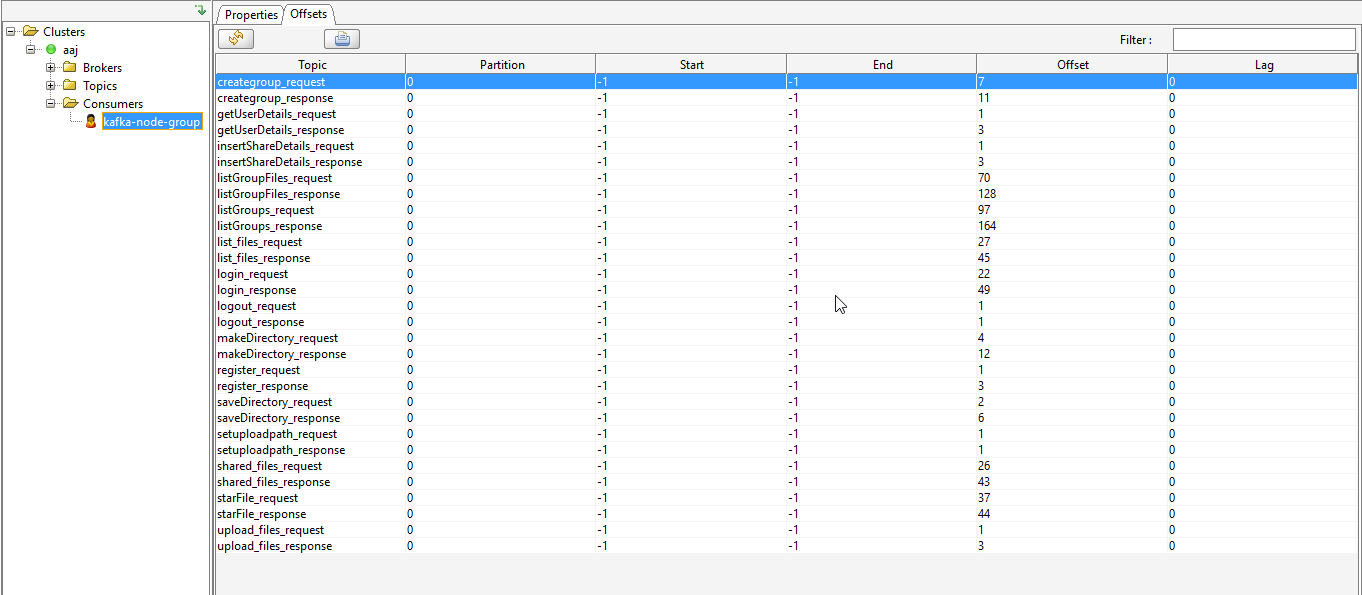
kafka-consumer

kafka-consumer

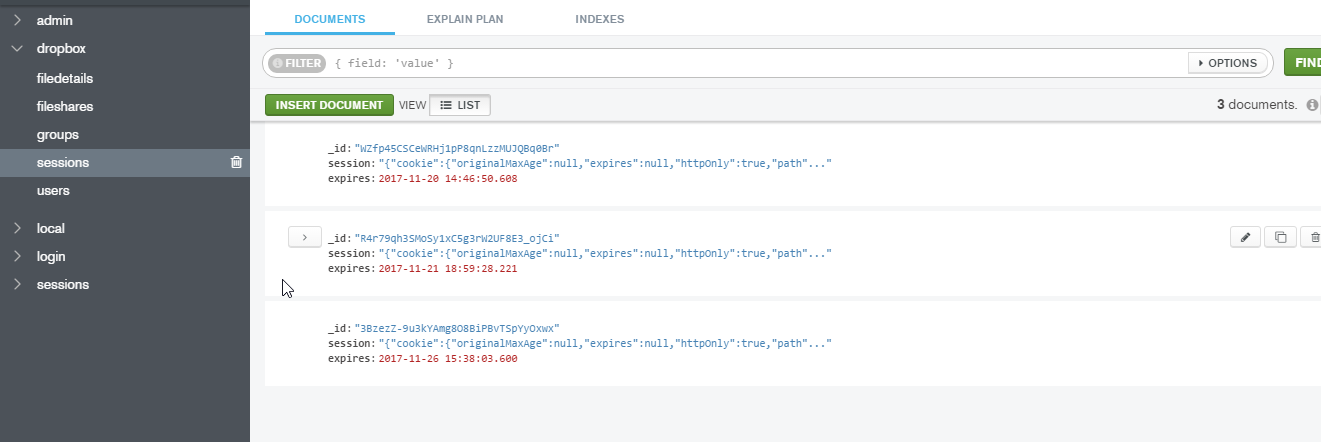
kafka-consumer

MongoDB

### Kafka-Topics



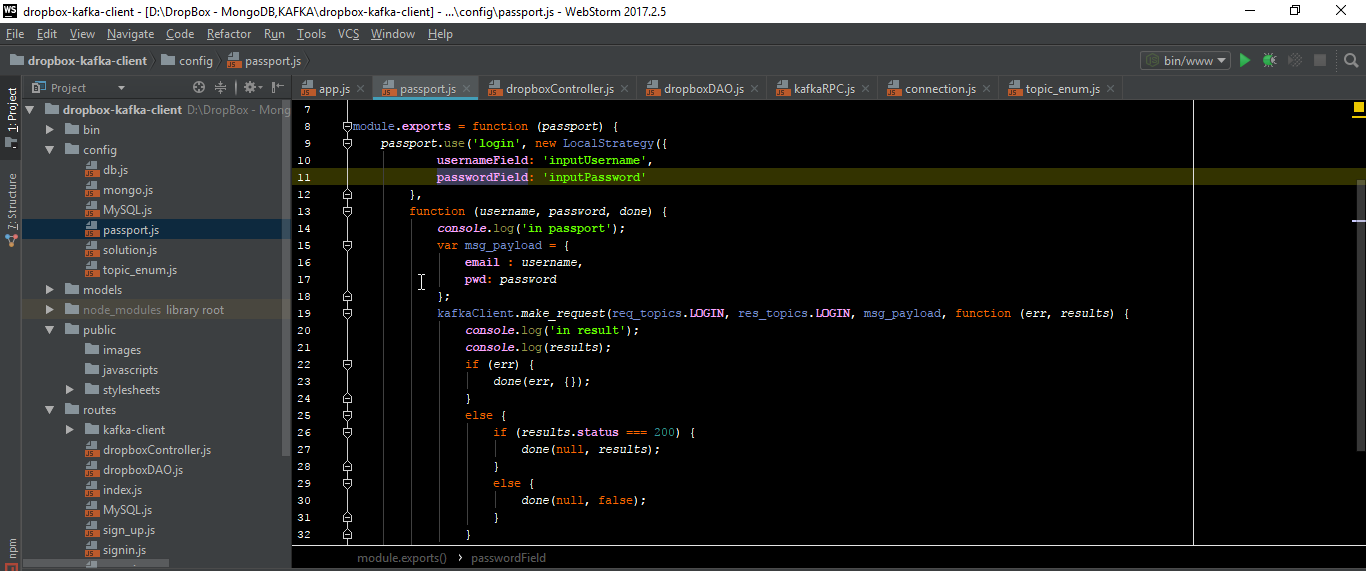
### MongoDB for Storing sessions



**Configuration in app.js file of dropbox-kafka-client for storing sessions**

var express = *require*(**'express'**);  
var passport = *require*(**'passport'**);  
*require*(**'./config/passport'**)(passport);  
  
var path = *require*(**'path'**);  
var favicon = *require*(**'serve-favicon'**);  
var logger = *require*(**'morgan'**);  
var cookieParser = *require*(**'cookie-parser'**);  
var bodyParser = *require*(**'body-parser'**);  
var index = *require*(**'./routes/index'**);  
var dropboxController = *require*(**'./routes/dropboxController'**);  
var session = *require*(**'client-sessions'**);  
var cors = *require*(**"cors"**);  
var multer = *require*(**"multer"**);  
var db = *require*(**'./config/db'**);  
var app = express();  
  
var mongoSessionURL = **"mongodb://localhost:27017/dropbox"**;  
var expressSessions = *require*(**"express-session"**);  
var mongoStore = *require*(**"connect-mongo"**)(expressSessions);  
  
var corsOptions = {  
 **origin**: **'http://localhost:3000'**,  
 **credentials**: true,  
 **optionsSuccessStatus**: 200 // some legacy browsers (IE11, various SmartTVs) choke on 204  
};  
  
  
//Enable CORS  
app.use(cors(corsOptions));  
  
// view engine setup  
app.set(**'views'**, path.join(\_\_dirname, **'views'**));  
app.set(**'view engine'**, **'ejs'**);  
  
// uncomment after placing your favicon in /public  
//app.use(favicon(path.join(\_\_dirname, 'public', 'favicon.ico')));  
app.use(logger(**'dev'**));  
app.use(expressSessions({  
 **secret**: **"CMPE273passport"**,  
 **resave**: false,  
 //Forces the session to be saved back to the session store, even if the session was never modified during the request  
 **saveUninitialized**: false, //force to save uninitialized session to db.  
 //A session is uninitialized when it is new but not modified.  
 **duration**: 30 \* 60 \* 1000,  
 **activeDuration**: 5 \* 60 \* 1000,  
 **store**: new mongoStore({  
 **url**: mongoSessionURL  
 })  
}));  
app.use(passport.initialize());

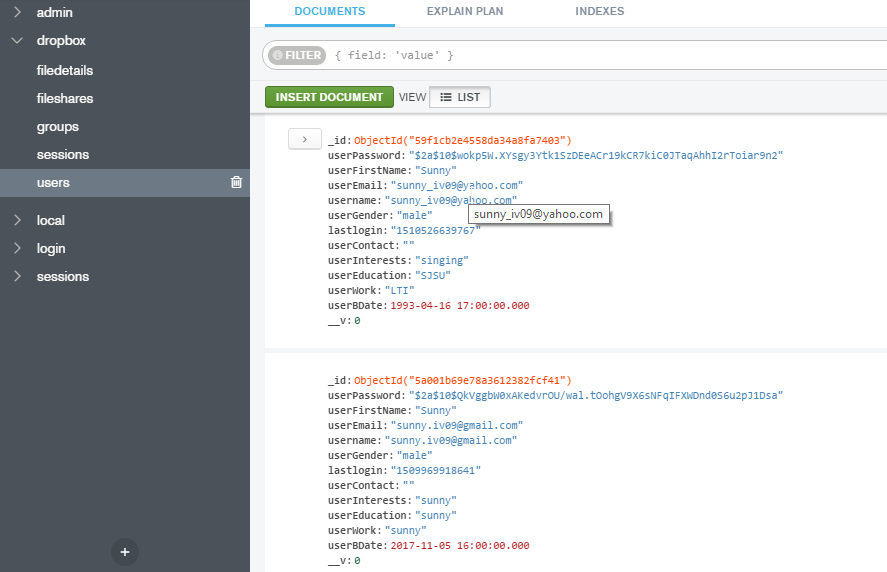
### PassportJS for Authentication



**PassportJS configuration in config/passport.js file of dropbox-kafka-client**

var passport = *require*(**'passport'**);  
var LocalStrategy = *require*(**'passport-local'**).Strategy;  
  
var req\_topics = *require*(**"topic\_enum"**).**req\_topic\_names**;  
var res\_topics = *require*(**"topic\_enum"**).**res\_topic\_names**;  
var kafkaClient = *require*(**'../routes/kafka-client/kafkaClient'**);  
  
module.**exports** = function (*passport*) {  
 *passport*.use(**'login'**, new LocalStrategy({  
 **usernameField**: **'inputUsername'**,  
 **passwordField**: **'inputPassword'** },  
 function (*username*, *password*, *done*) {  
 ***console***.log(**'in passport'**);  
 var msg\_payload = {  
 **email**: *username*,  
 **pwd**: *password* };  
 kafkaClient.*make\_request*(req\_topics.**LOGIN**, res\_topics.**LOGIN**, msg\_payload, function (*err*, *results*) {  
 ***console***.log(**'in result'**);  
 ***console***.log(*results*);  
 if (*err*) {  
 *done*(*err*, {});  
 }  
 else {  
 if (*results*.**status** === 200) {  
 *done*(null, *results*);  
 }  
 else {  
 *done*(null, false);  
 }  
 }  
 });  
 })  
 )  
};

### Password Encryption



**Configuration in dropboxController.js for encryption as soon as password is received**

exports.registerUser = function (*req*, *res*) {  
  
 var id = *req*.param(**"userEmail"**);  
 var pwd = bcrypt.*hashSync*(*req*.param(**"password"**));  
 var fn = *req*.param(**"firstName"**);  
 var ln = *req*.param(**"lastName"**);  
 var bdate = *req*.param(**"dob"**);  
 var gender = *req*.param(**"gender"**);  
 var edu = *req*.param(**"edu"**);  
 var work = *req*.param(**"work"**);  
 var inter = *req*.param(**"inter"**);

### Connection Pooling for mongodb connections

**Configuration in config/mongo.js in dropbox-kafka-server for connection pooling**

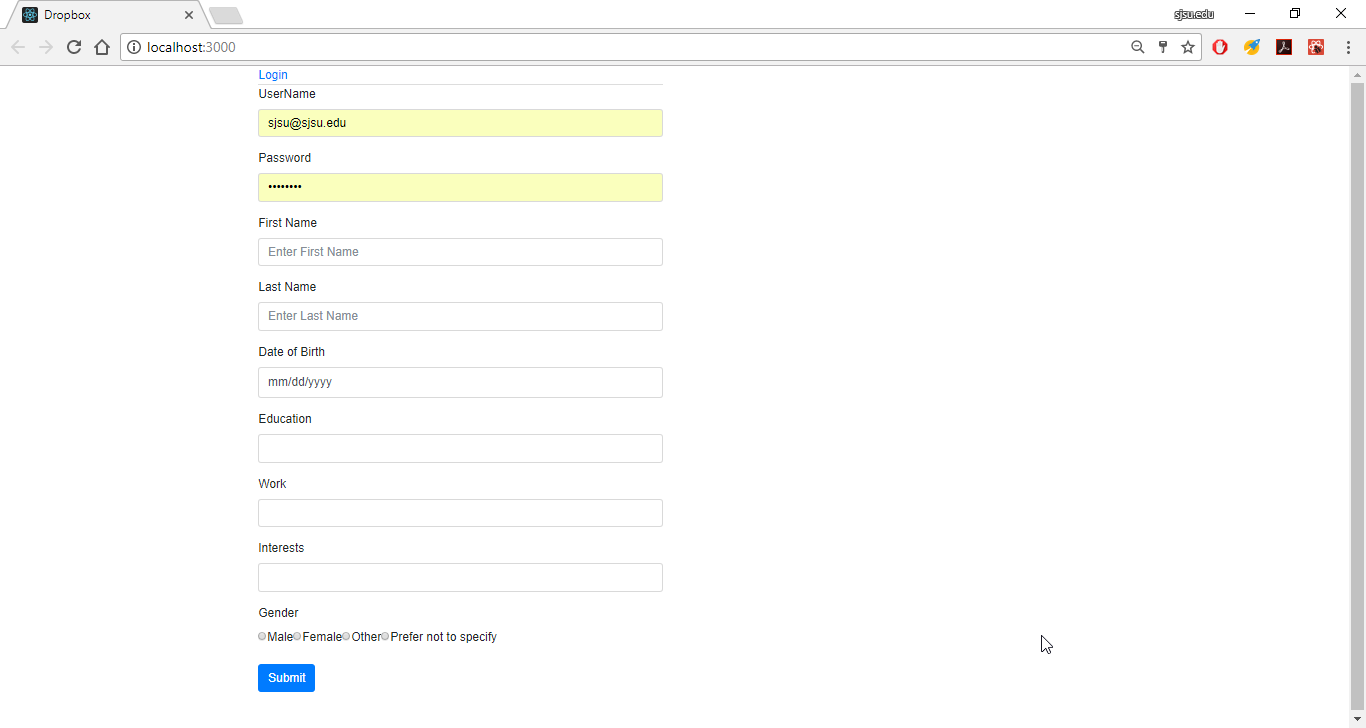
var poolSize = 10;  
var queueSize = 10;  
  
var pool = [];  
var queue = [];  
var queueCount = 0;  
  
var queueNotifier = new **Map**();  
  
  
function *CreateConnectionPool*() {  
  
 for (var i = 0; i < poolSize; i++) {  
 var connection = ***mongoose***.connect(dbURI);  
 pool.push(connection);  
 }  
 ***console***.log(pool[0]);  
 return this;  
}  
  
function *getConnection*(*callback*) {  
  
 ***console***.log(**"connection requested"**);  
  
 if (*isConnectionFree*()) {  
  
 ***console***.log(**"connection free"**);  
 *callback*(pool.pop());  
  
  
 } else {  
  
 ***console***.log(**"connection not free"**);  
 if (*isQueueFree*()) {  
  
 ***console***.log(**'in queue'**);  
 queue.push(queueCount);  
 queueNotifier.set(queueCount, false);  
 **token** = queueCount;  
 queueCount++;  
 *waitInQueue*(**token**, function (*conn*) {  
 *callback*(*conn*)  
 });  
  
 } else {  
  
 ***console***.log(**'queue not free'**);  
 return null;  
 }  
 }  
}  
  
function *waitInQueue*(*token*, *callback*) {  
  
 while (!queueNotifier.get(*token*)) {  
  
 if (queueNotifier.get(*token*)) {  
 if (*isConnectionFree*()) {  
 ***console***.log(**'waiting'**);  
 // return (pool.pop());  
 *callback*(pool.pop());  
 }  
 }  
  
 }  
  
}  
  
function *releaseConnection*(*connection*) {  
  
 pool.push(*connection*);  
 ***console***.log(**'connection released'**);  
 queueNotifier.set(queue.pop(), true);  
 queue.shift();  
  
}  
  
function *isConnectionFree*() {  
  
 return pool.**length** > 0;  
  
}  
  
function *isQueueFree*() {  
  
 return queue.**length** < queueSize;  
}  
  
exports.*CreateConnectionPool* = *CreateConnectionPool*;  
exports.*getConnection* = *getConnection*;  
exports.*releaseConnection* = *releaseConnection*;

**Different pages and their functionalities:**

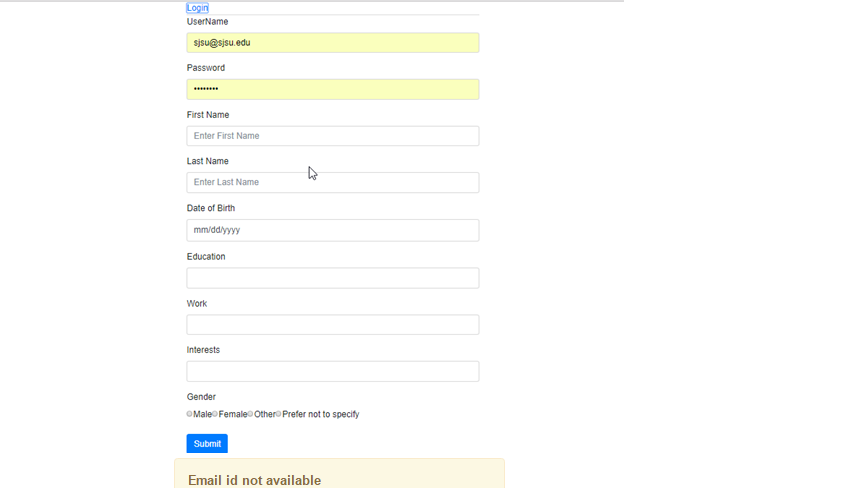
1. **Signup page**: Allows the user to sign up for the application so that they can use it to share and upload their files. It takes a variety of user input to provide a personalized experience on login.
2. **Login page**: Allows the user to login to the application and create user session. The user gets redirected to the home page on successful login or shown a validation message on incorrect inputs.
3. **Home page**: It serves multiple functionalities of listing user’s uploaded and starred files. It also shows the files/folders shared by others with the user. Also allows the user to upload multiple files and create folders.
4. **Profile page**: Shows details regarding user profile and user activity.
5. **Group page**: Show user’s groups, files shared in the group, members of group. Based on group ownership delete group member functionality.

## **Results:**

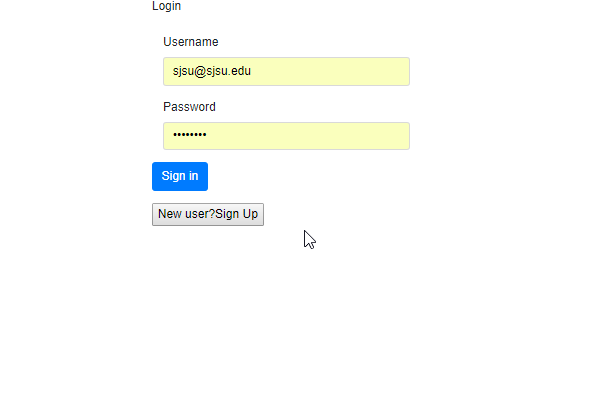
**Screen captures of dropbox prototype:**

**Register**: 

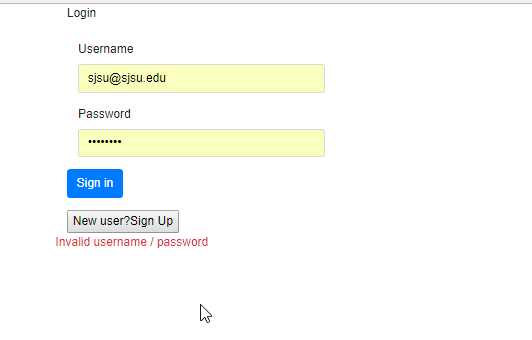
**Validation**: If user tries to register with already existing email-address, it will display message as show below:



**Login:**

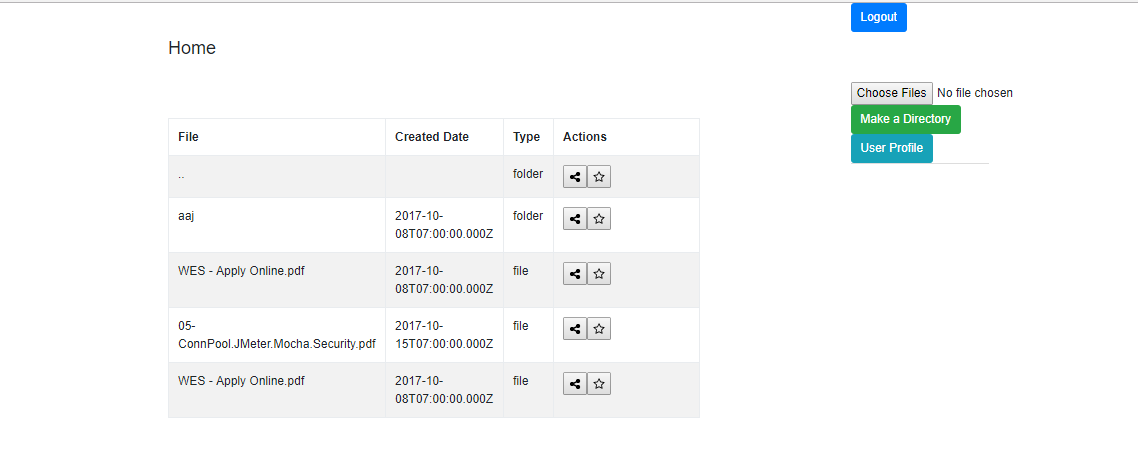
****

**Validation: If incorrect username or password**

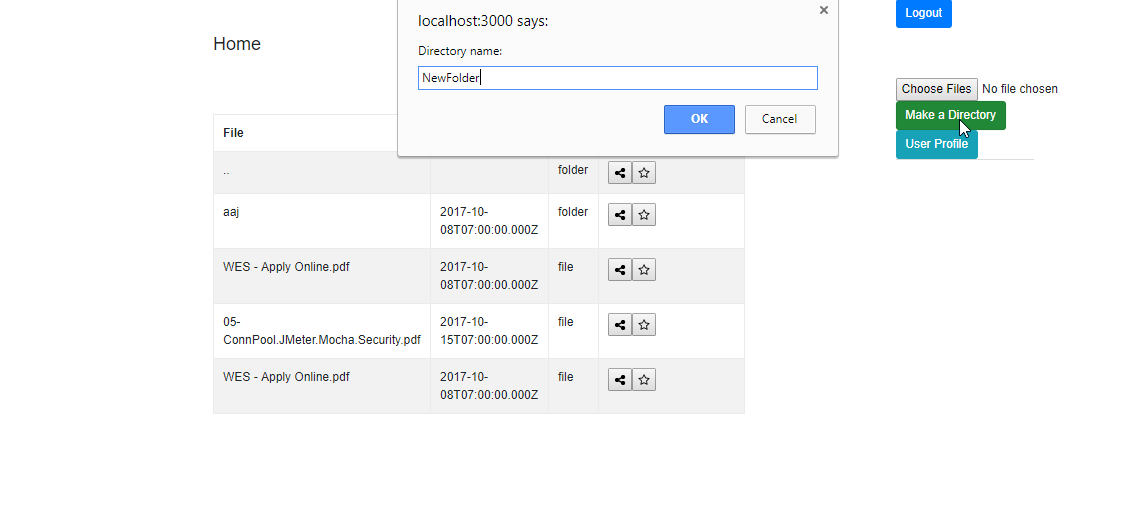
****

**Home:**

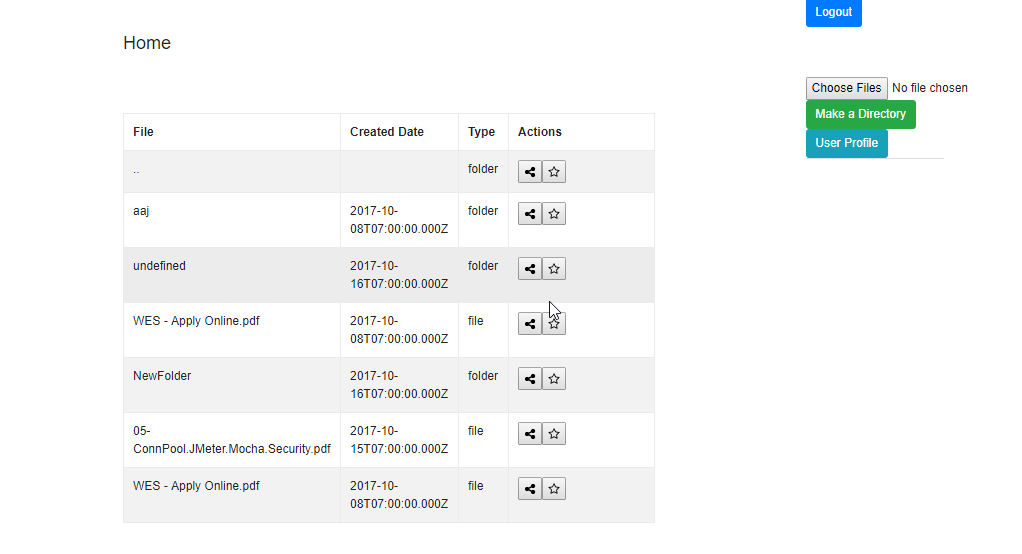
User gets to the home page after successful login. The home page shows a list of files that are uploaded and starred by user. It also has options to logout of system, create a directory or to view user profile.



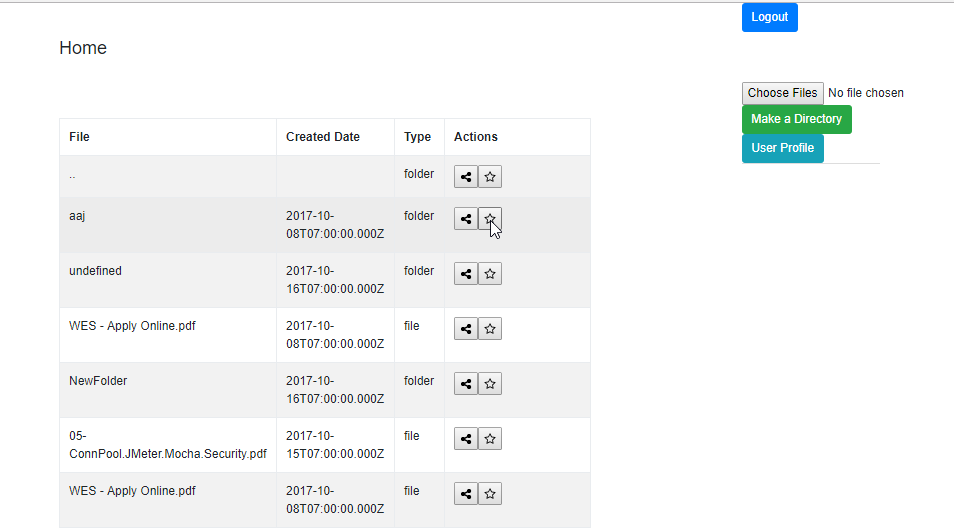
**Create a directory:**

****

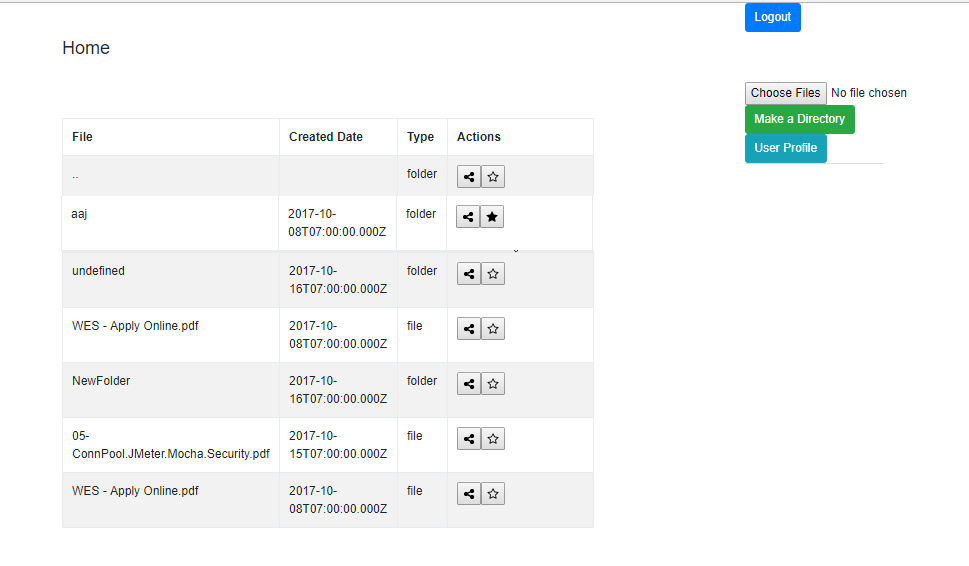
**Result:**

******

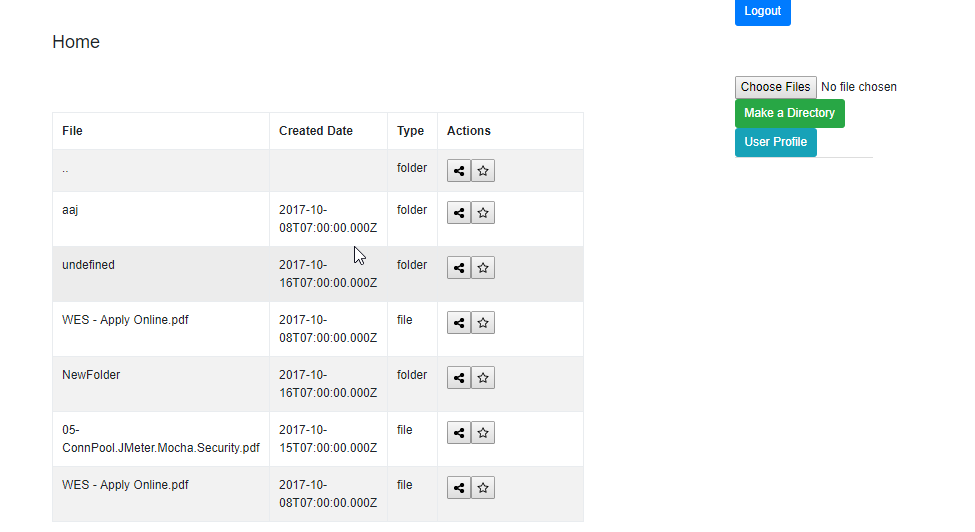
**Star a directory:**

****

**Result:**

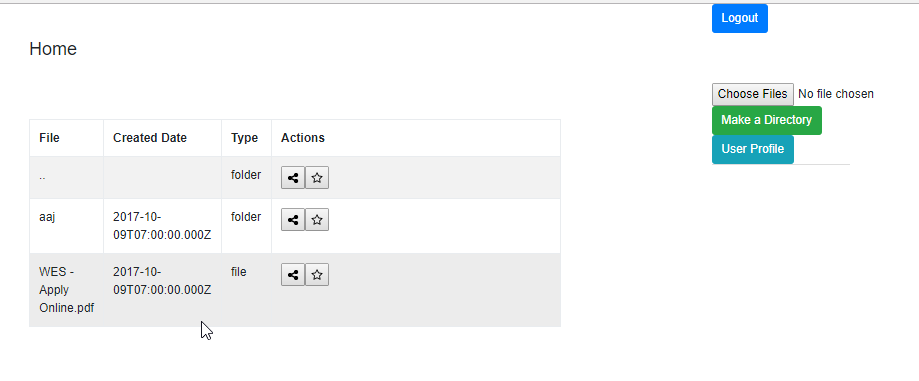
****

**Go inside Directory:**

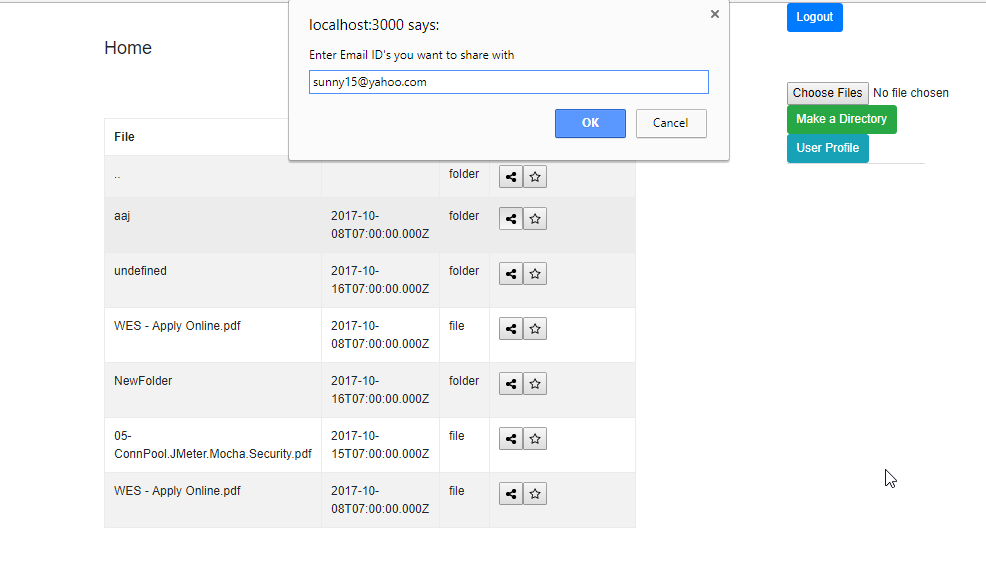
****

**Result:**

Lists files inside the directory. And allows user to upload files inside the directory.

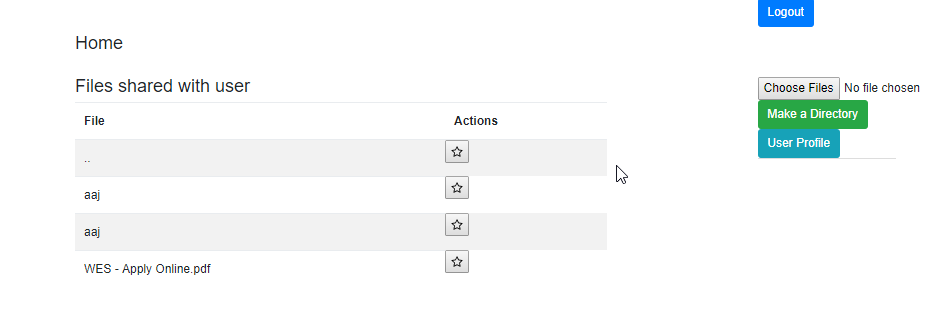
****

**Share file/directory**

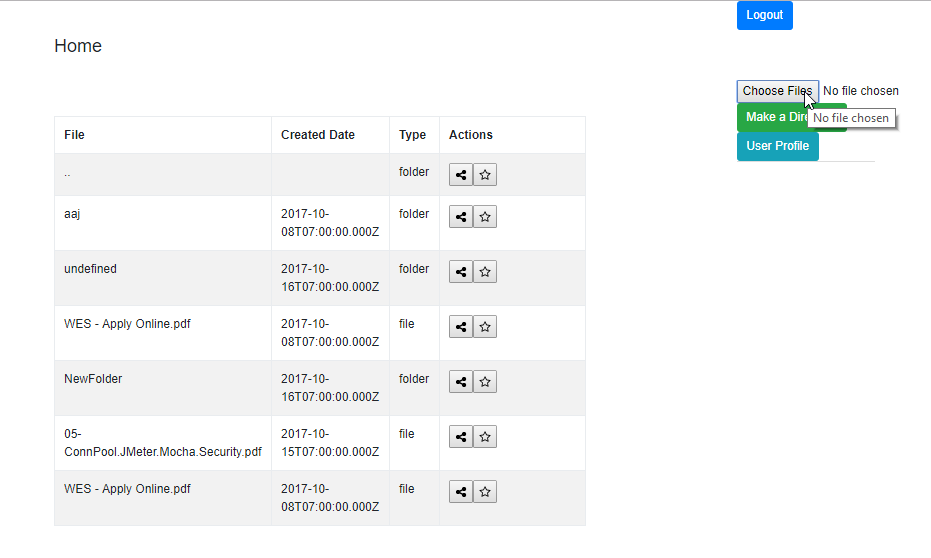
****

**Result:**

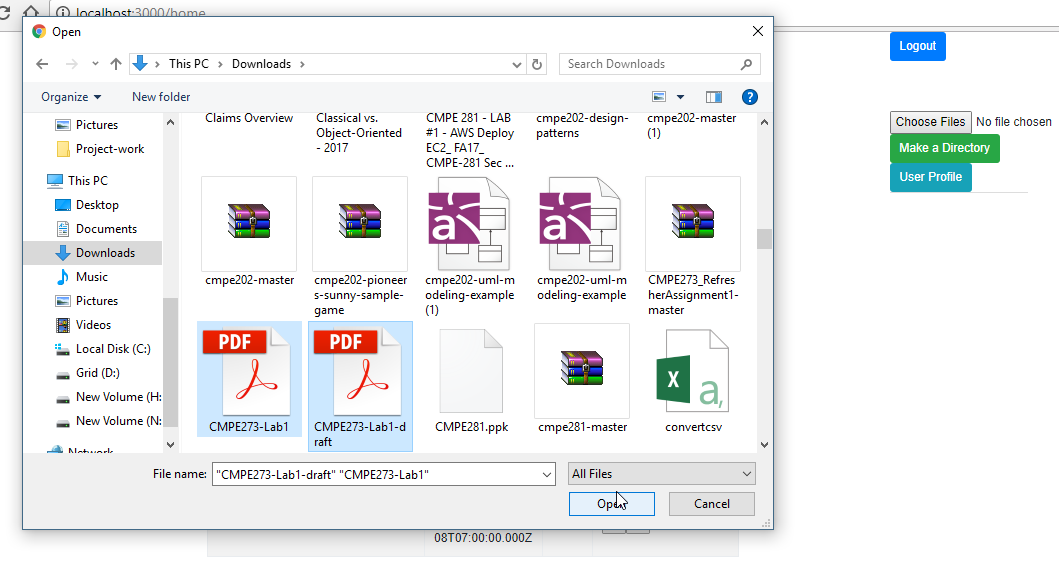
When I login with [sunny15@yahoo.com](mailto:sunny15@yahoo.com) I will be shown another table with files shared with the user.

****

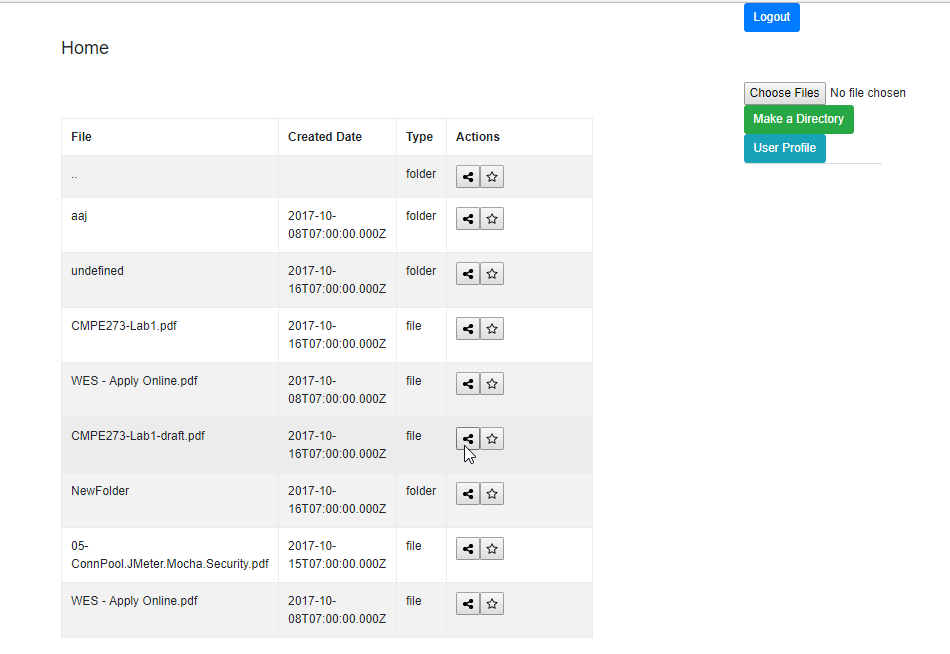
**Upload files:**

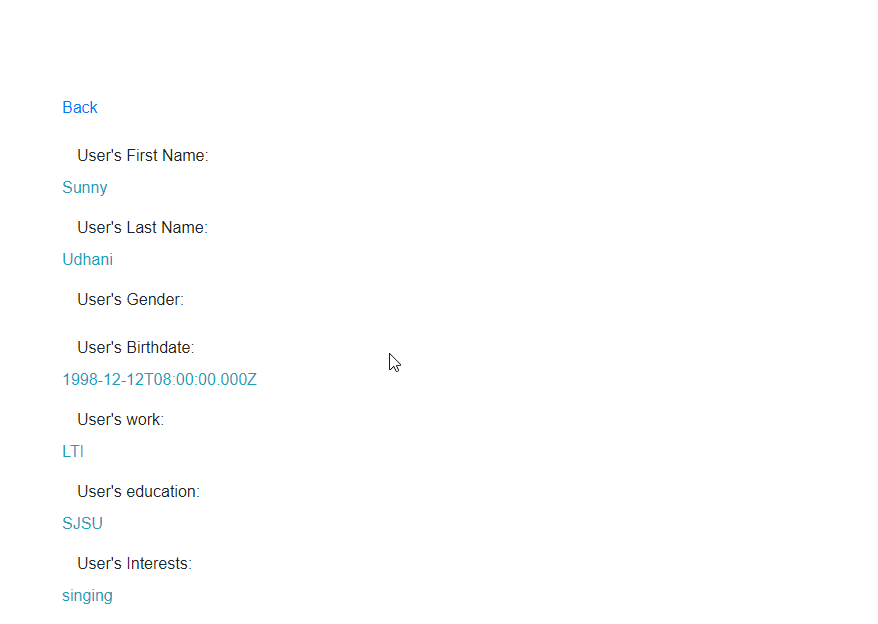
****

**Select files to upload :** two files selected for uploading.



**Result:**

****

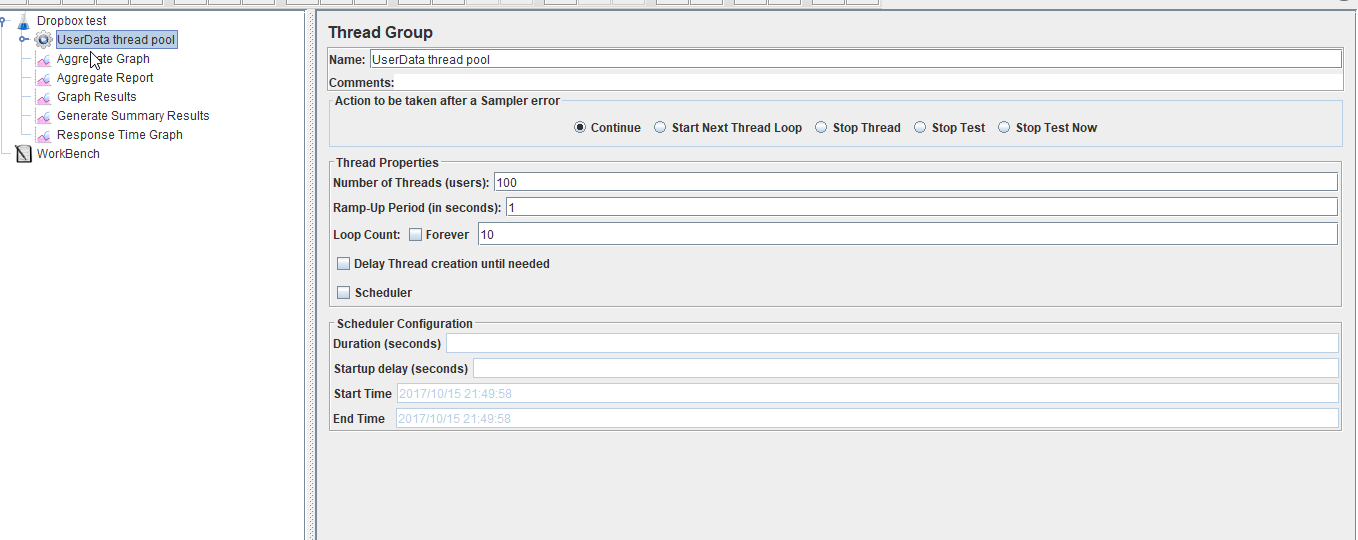
**User Profile:** Display the logged-in user’s details in a page ****

## **Performance**

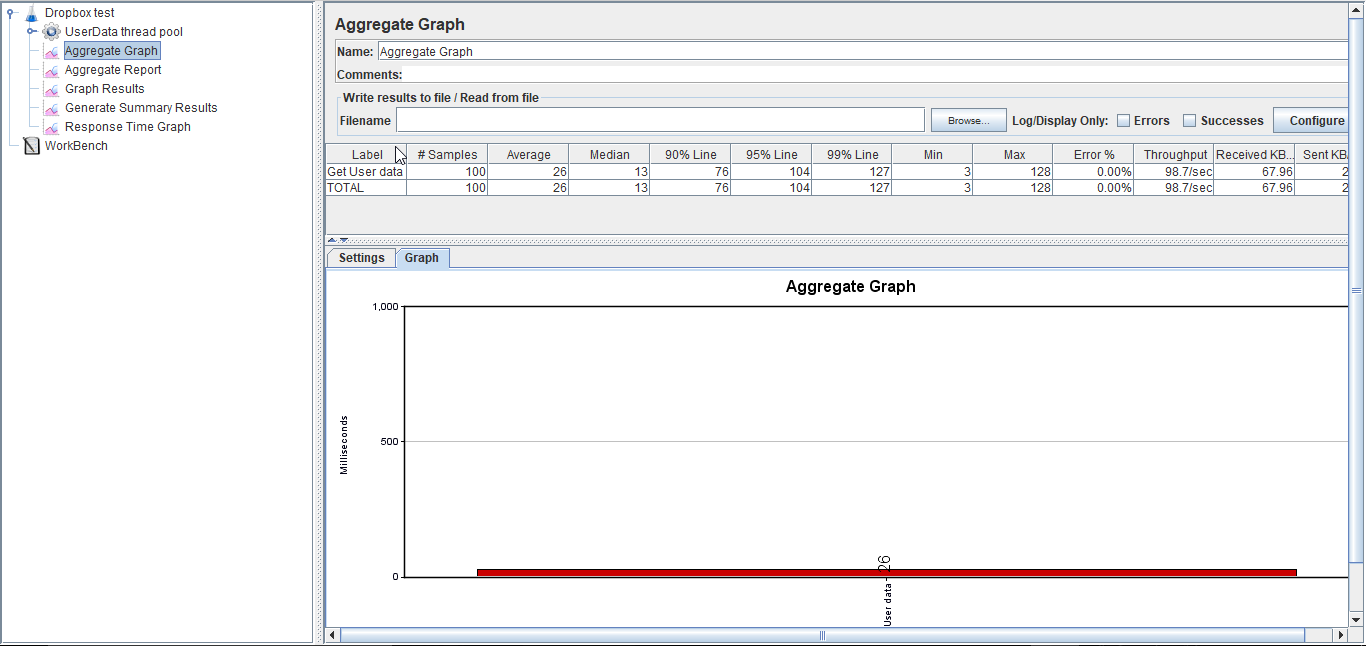
**JMETER TESTING**

1. For 100 concurrent users:

**Setup**-

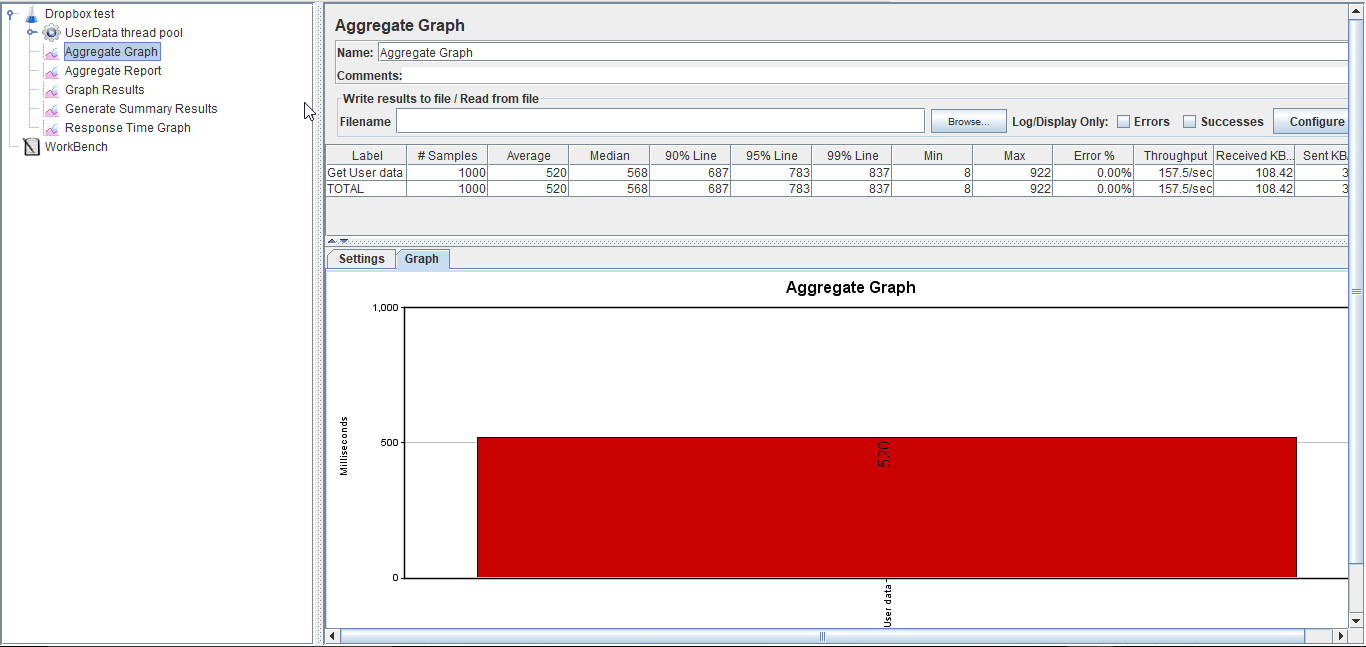


**With Connection Pooling**:



****

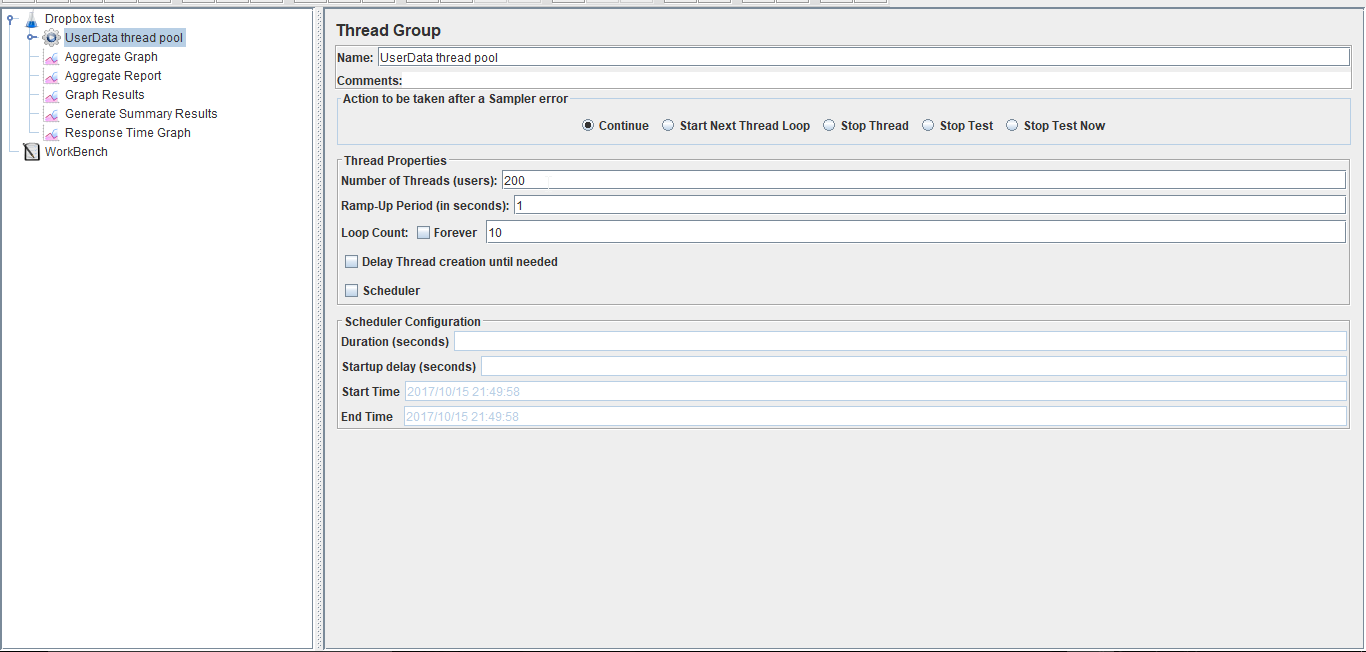
**Without Connection Pooling**:



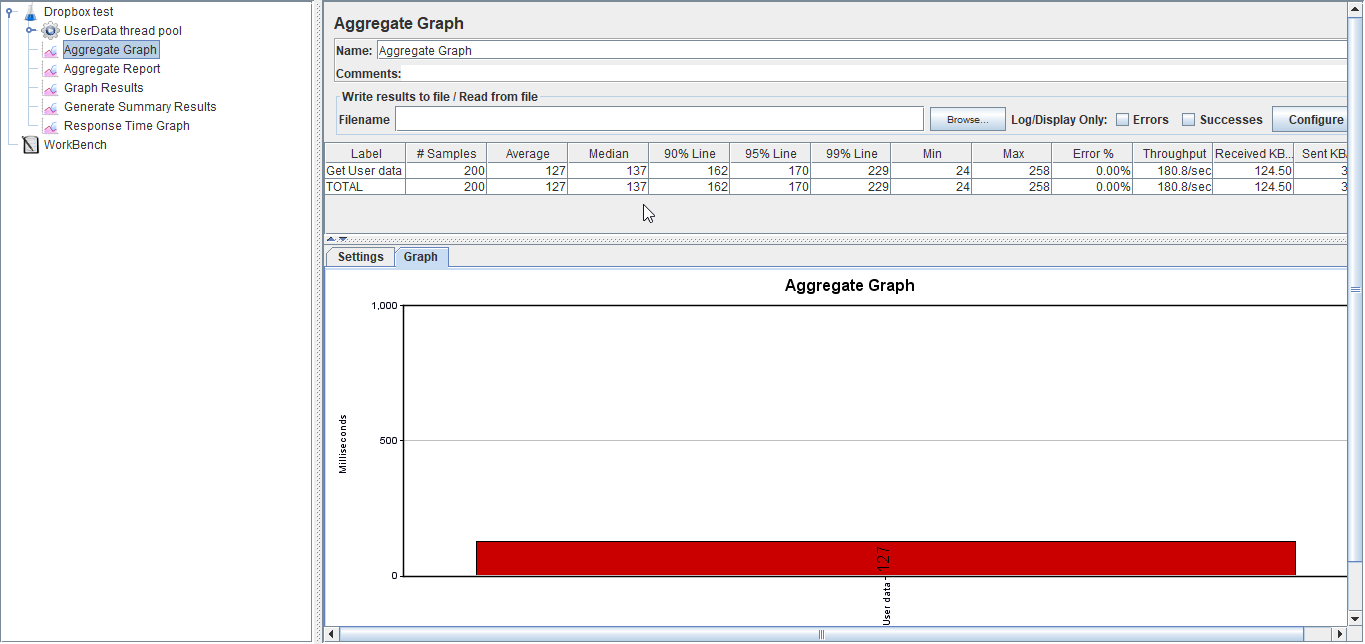
****

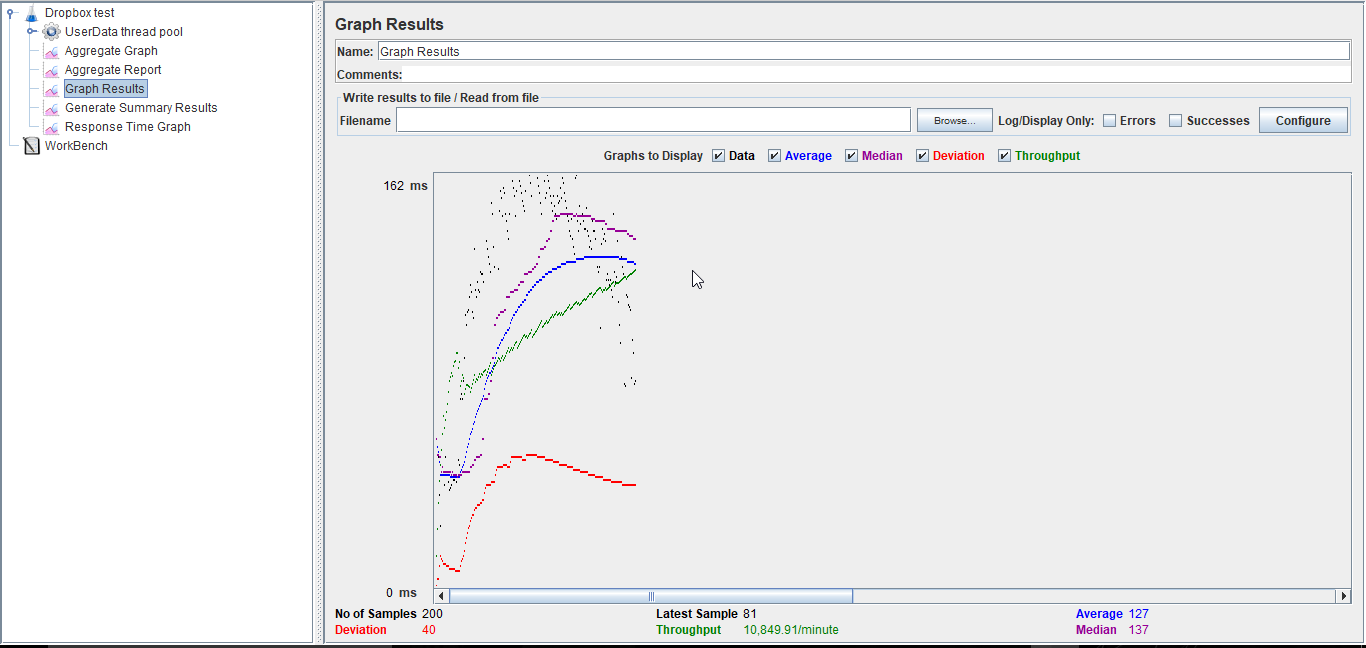
1. **For 200 concurrent users.**

Setup –

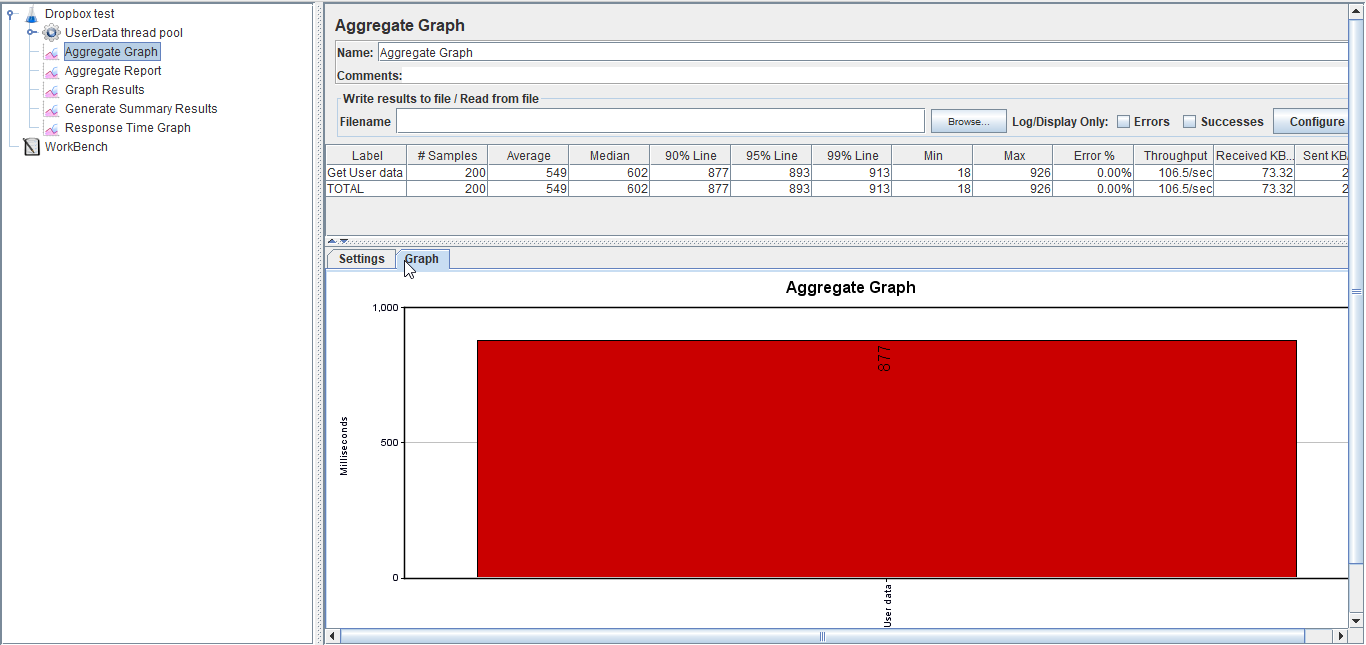


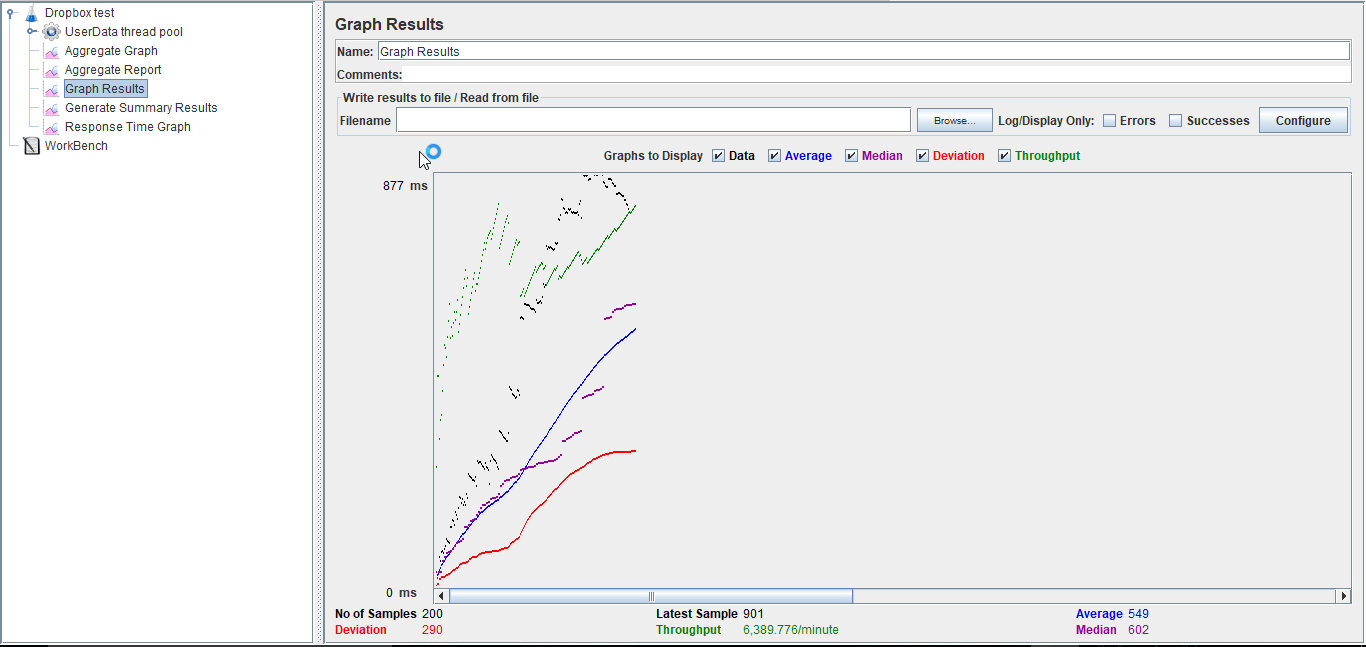
**With Connection Pooling**:





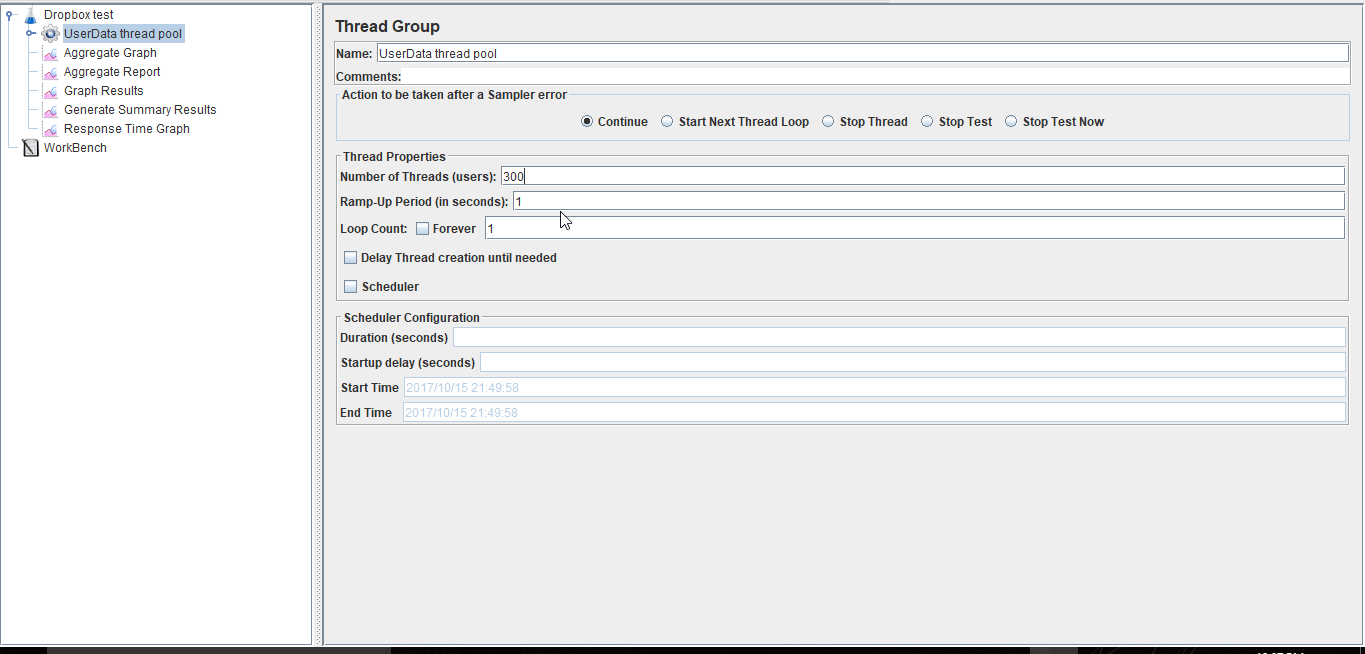
**Without connection pooling**:



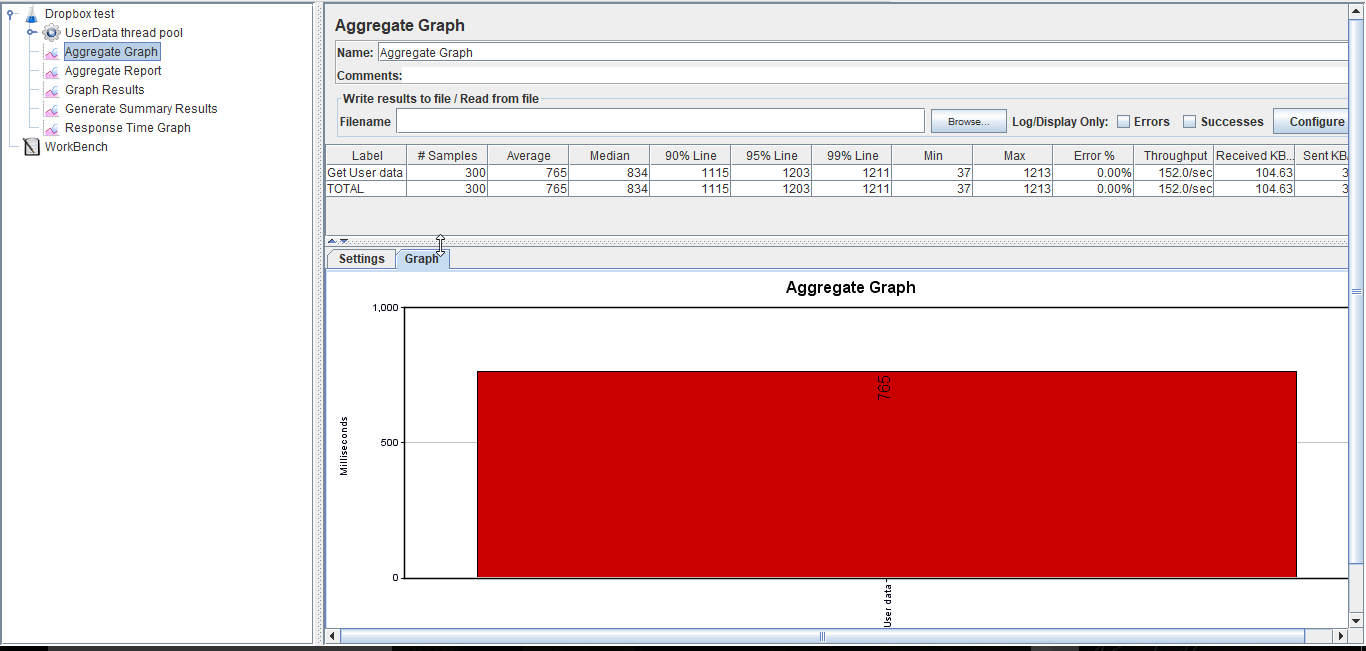


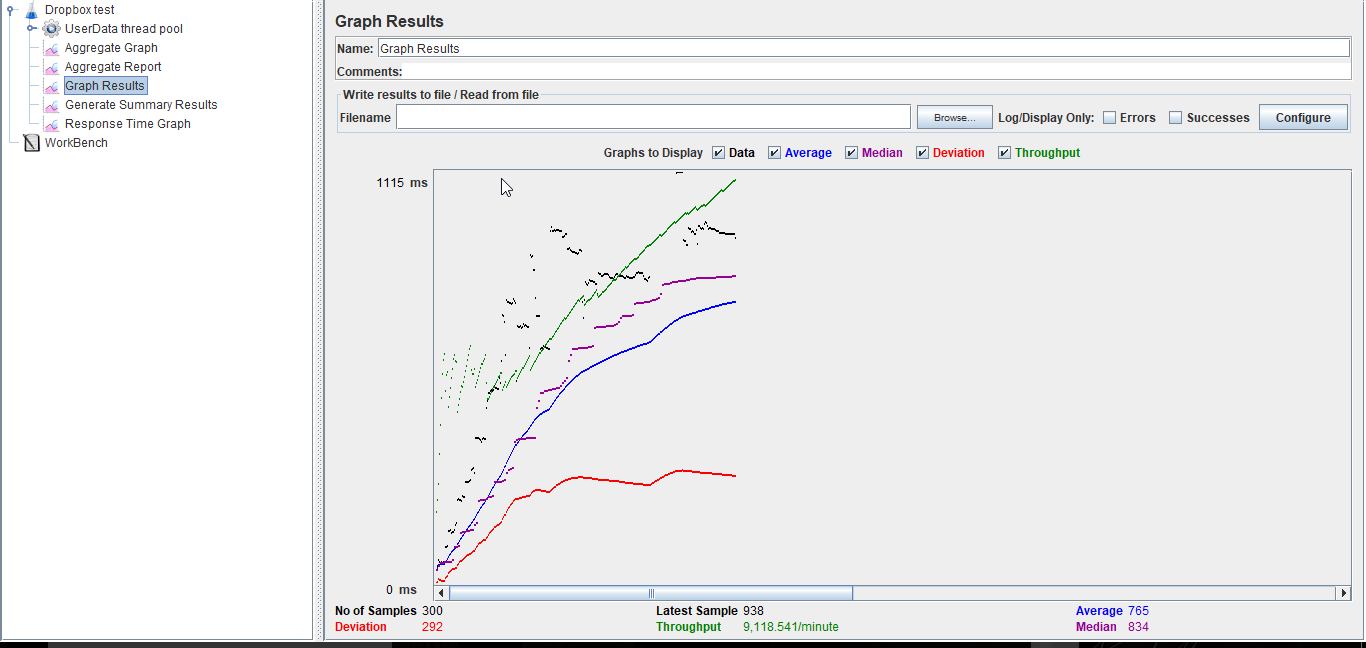
1. **For 300 concurrent users**:

**Setup** –

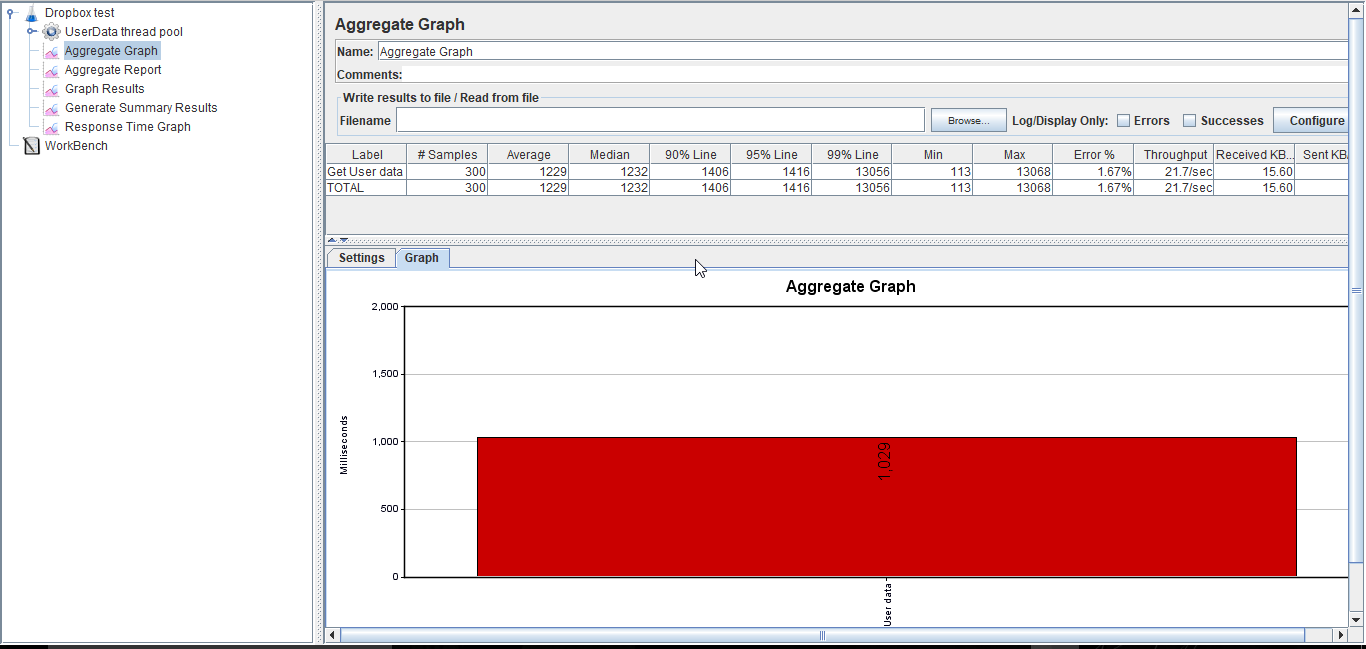


**With connection pooling**:





**Without Connection Pooling**:

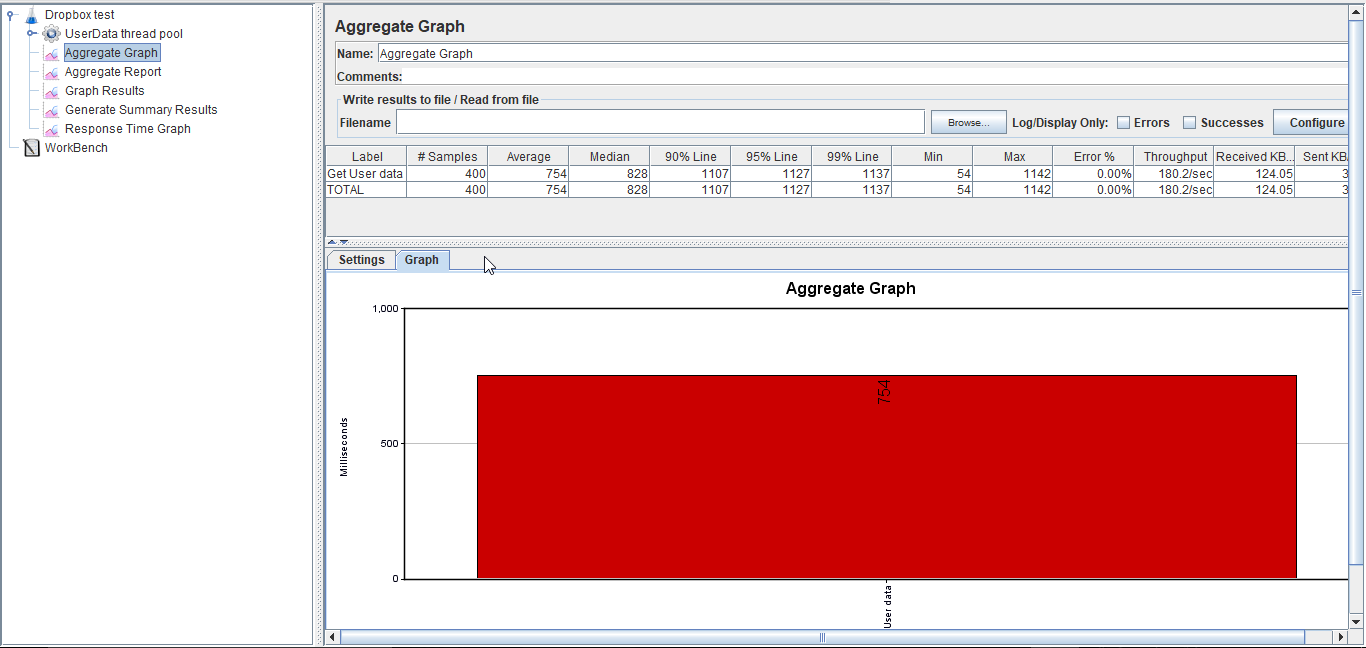




1. **For 400 concurrent users**:

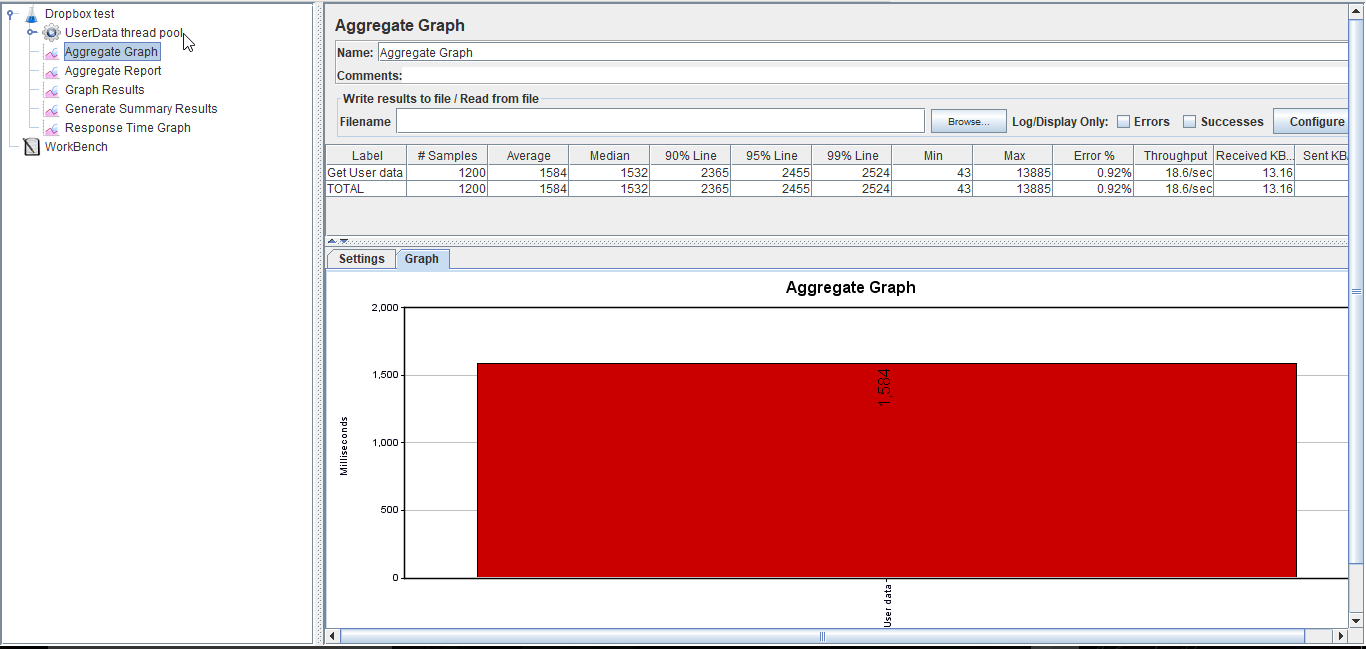
**Setup** –

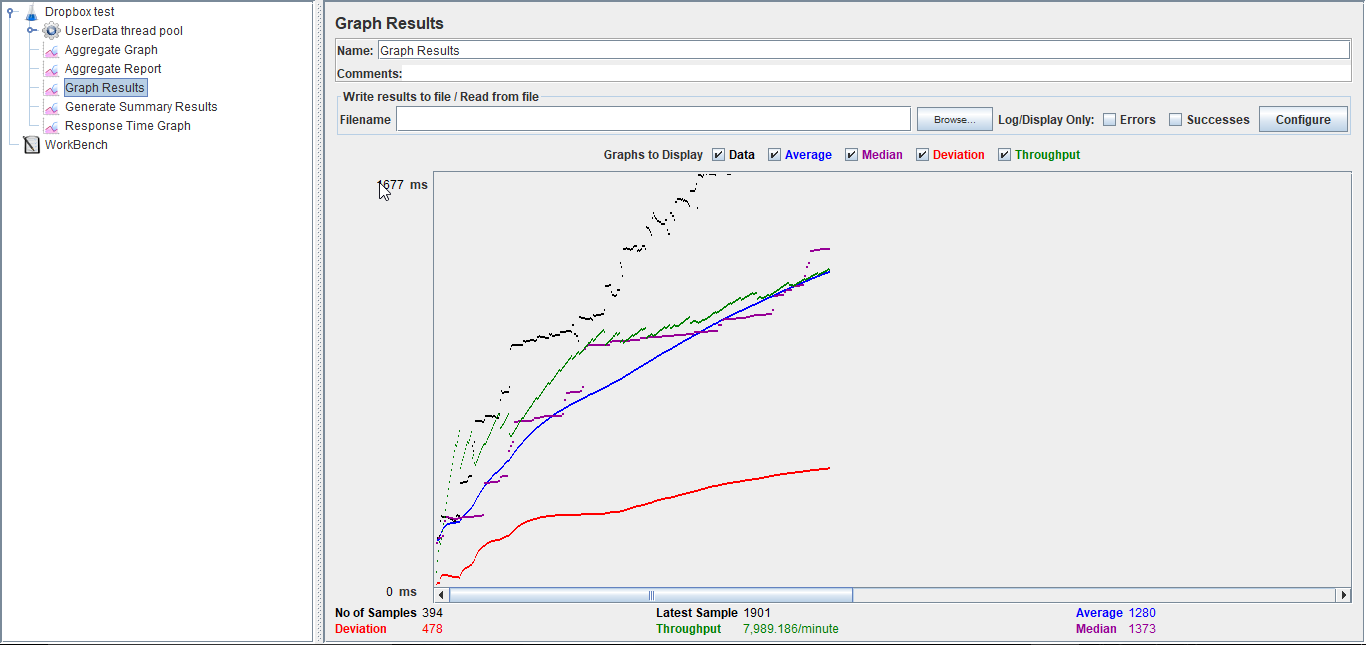
**With connection pooling**:





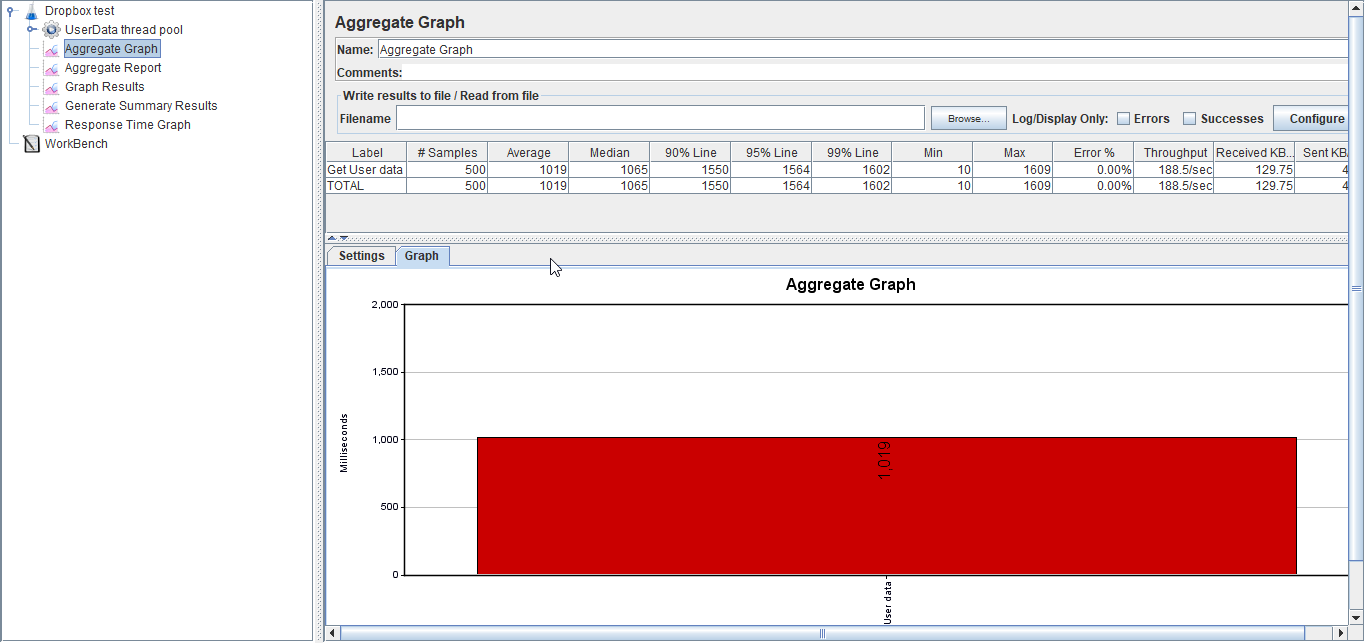
**Without Connection Pooling**:





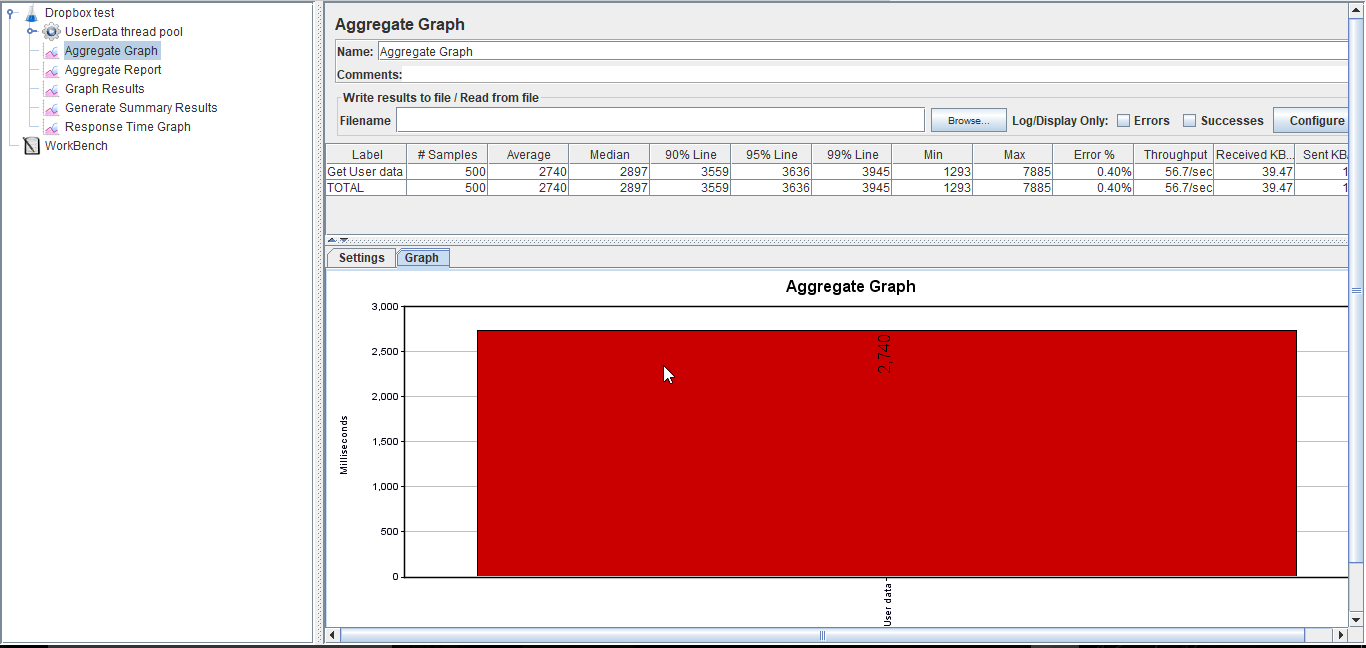
1. For 500 concurrent users:

**With Connection pooling**:



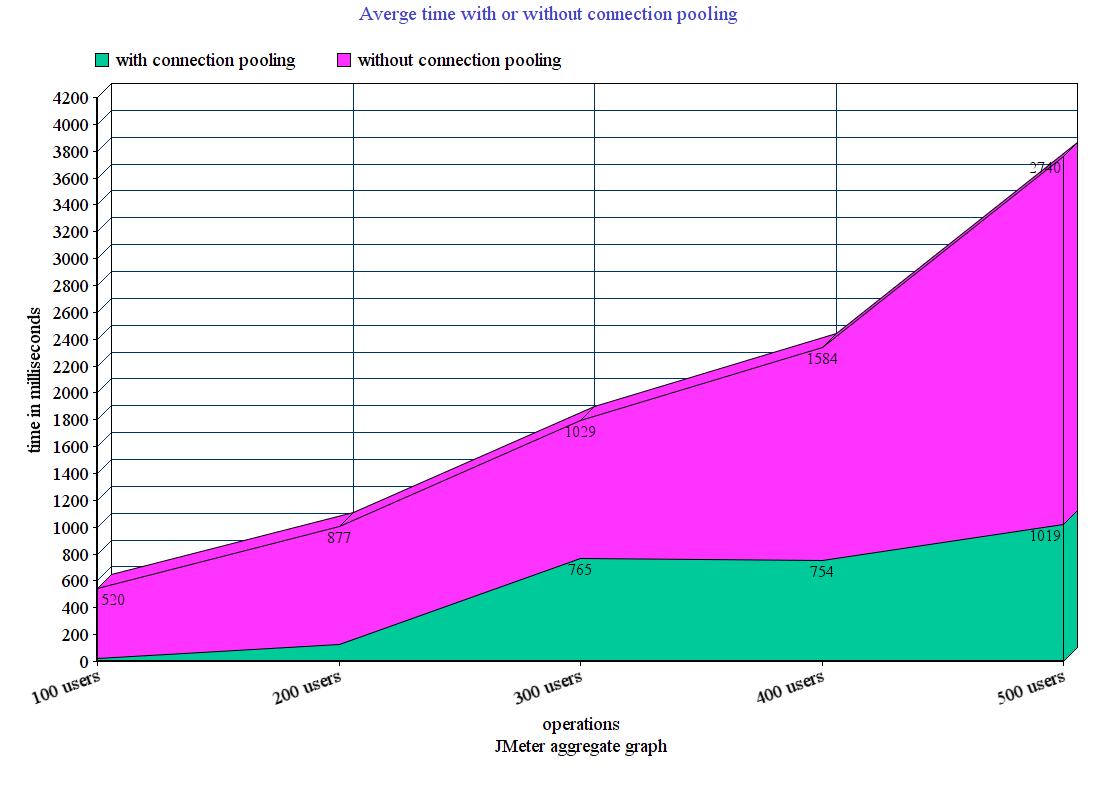


**Without Connection Pooling**:

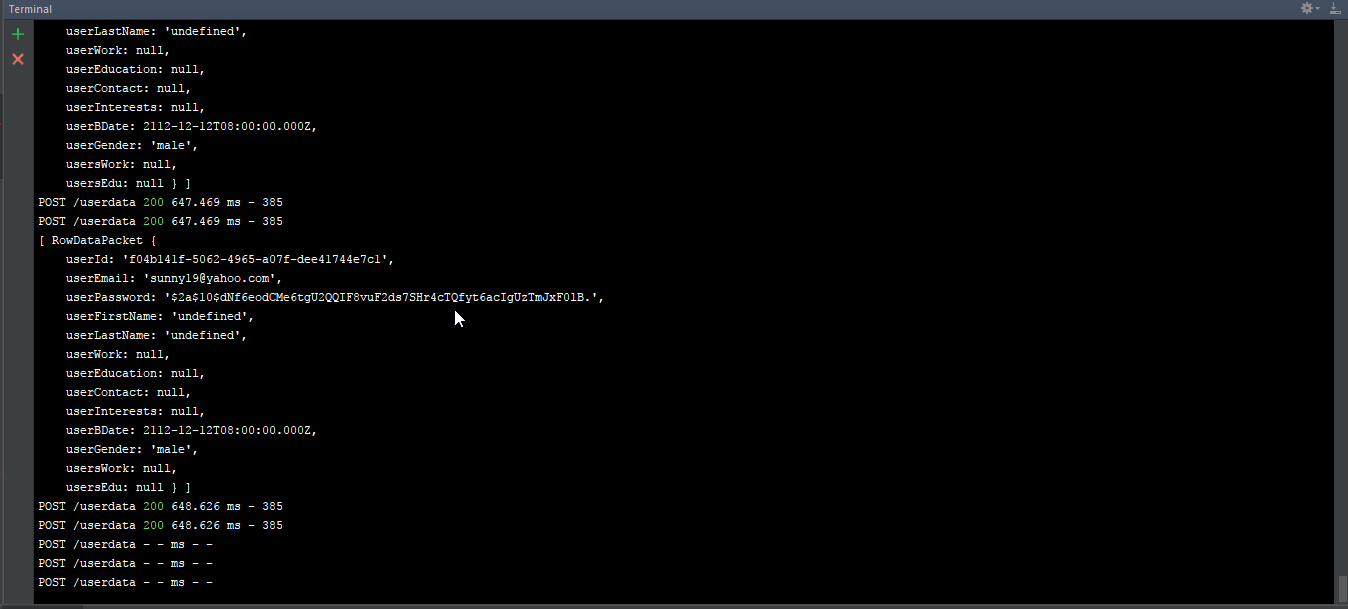




## **Graph showing average time for 100,200,300,400 and 500 concurrent users with and without connection pool.**



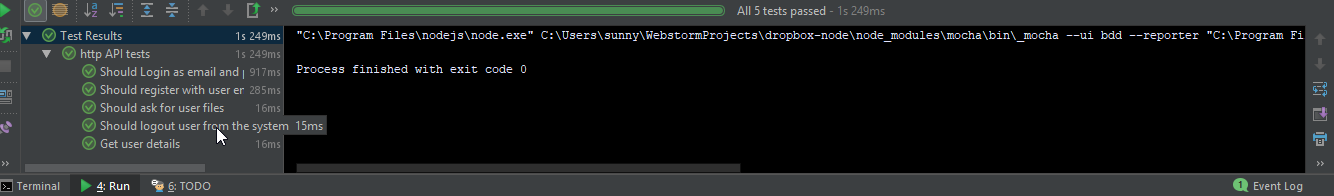
**Server:**

****

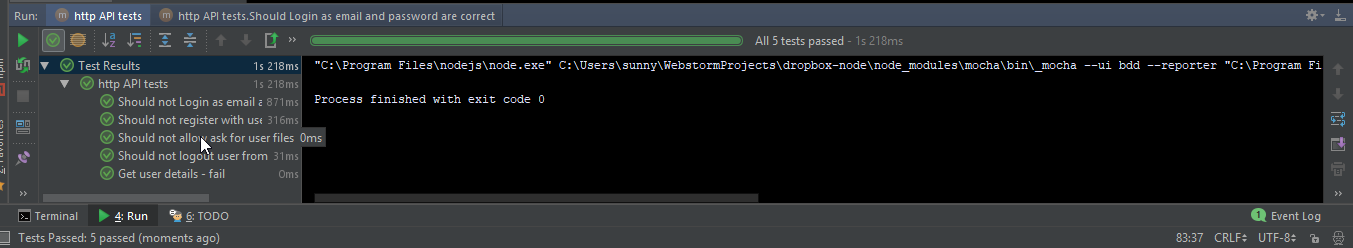
**MOCHA TESTING:**

I have tested 10 API calls using MOCHA.

**Positive Test cases:**



**Negative Test cases:**

****

# **Lab 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..*

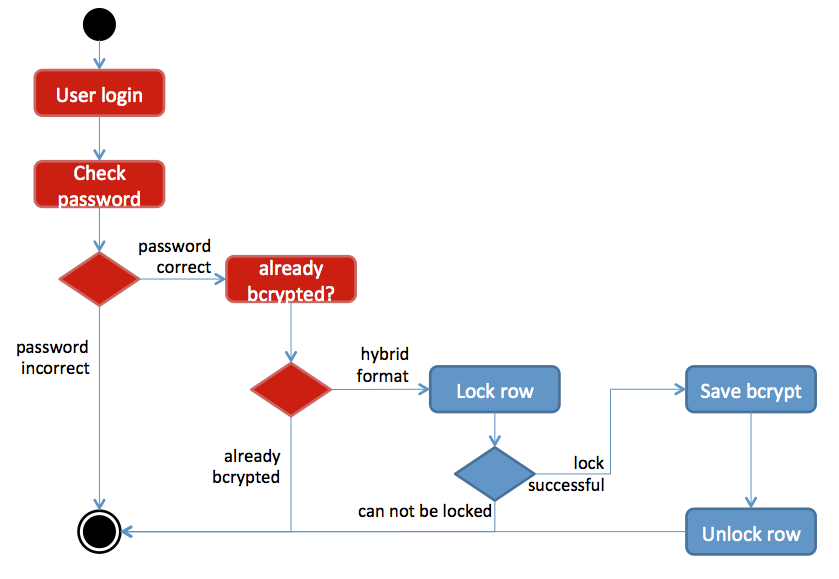
I have used nodejs library “bcrypt” which provides **bcrypt algorithm** for encrypting and validating user’s password. It is based on **blowfish cipher**. There are many other encrypting algorithms like: AES, SHA (1, 224, 256, 384, 512), RSA, TRIPLE DES, Twofish etc.

I selected bcrypt algorithm because of its expensive key setup phase. It starts off with a set of subkeys in standard state, which is used later to perform block encryption using part of key and use it to replace some of the subkeys. It repeats this process till it replaces all the subkeys Moreover, bcrypt is more resistant to brute force attacks because it is an adaptive function. Due to its adaptive nature, whenever a hacker tries to use brute-force attack to get into the system, overtime the iteration count keeps on increasing which makes it slower.

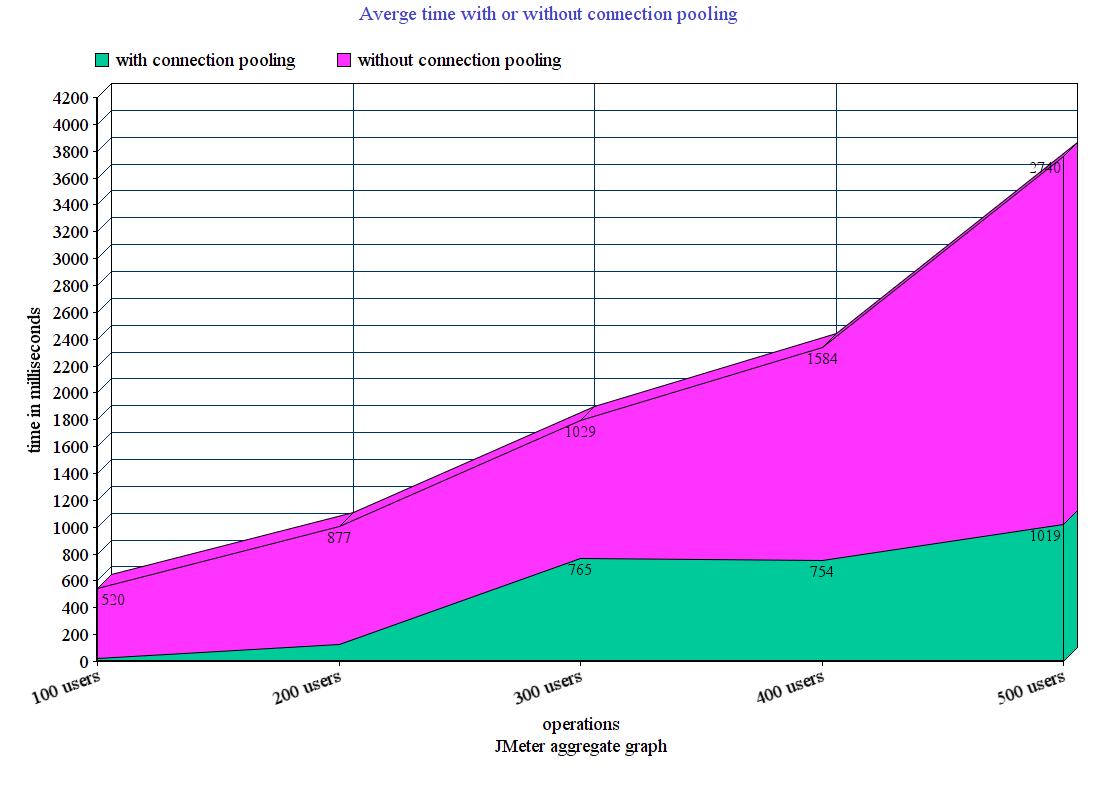
Bcrypt also adds random data that is used as an additional input. The random data added is called SALT. This also prevents any rainbow table attacks.

**Security Concern – Bcrypt algorithm only considers first 72 characters of a string and the rest are ignored.**

Bcrypt algorithm working:



1. *Compare the Results of the graphs with and without connection pooling. Explain the results in detail. Describe the algorithm of connection pooling used in your application.*



The graph above shows the comparison between the applications’ average response time when connection pooling is implemented and when it is not implemented.

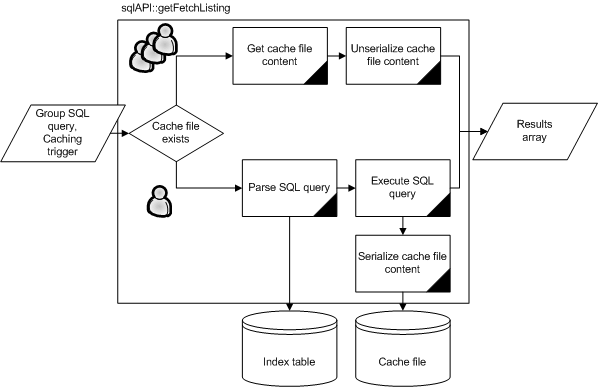
When there are 500 concurrent users each having one call, the average response time is 2740 millieseconds when no connection pooling is implemented, which reduces to 1019 millieseconds on implementing connection pooling. The reason for this significant drop in response time by 50% is that the connections are created when the application is started and are allocated from the pool to a request as and when a request comes, as a result the time of creating a new connection is reduced. The server picks up a connection from pool of connections and serves the request instead of wasting time in creating one.

In my application, initially 100 database connections will be created and added to the pool when the server starts where the maximum size of pool is 1000. When a user will access the server, he will get one connection from the pool (i.e. from array). When the connection ends, connection will be added back to the pool (i.e. it will be added back to the array).

var pool = mysql.createPool({  
 **connectionLimit** : 100,  
 **queueLimit** : 10,  
 **host** : **'localhost'**,  
 **user** : **'sunny'**,  
 **password** : **'sunny'**,  
 **database** : **'dropbox'**,  
 **port** : 3306  
});  
  
function *fetchData*(*callback*,*sqlQuery*){  
  
 ***console***.log(**"**\n**SQL Query : "**+*sqlQuery*);  
  
 pool.getConnection(function (*err* , *connection*) {  
 if(*err*){  
 ***console***.log(**"ERROR: "** + *err*.**message**);  
 *connection*.**release**();  
 }else {  
 *connection*.**query**(*sqlQuery*, function (*err*, *rows*, *fields*) {  
 if (*err*) {  
 ***console***.log(**"ERROR: "** + *err*.**message**);  
 }  
 else { // return err or result  
 *callback*(*err*, *rows*);  
 }  
 });  
 *connection*.**release**();  
 }  
 });  
}

1. *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.*

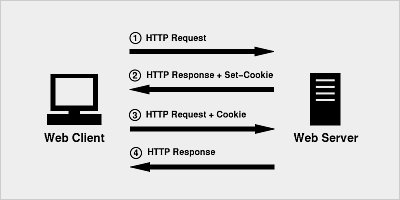
Caching in Databases:



As seen in the image above, the sql query execution plan or sql query is cached by the database system so as to decrease the amount of time it takes to execute a query. If a server sends the same query repeatedly to retrieve user details multiple times, the database will cache the execution plans for the query and when the query comes again the database doesn’t need to evaluate different plans again for executing the query and will immediately use the cache to respond to the execution request.

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

Session cookies

**

Yes, my session strategy is horizontally scalable. As I have used client sessions for session management, I save the session in cookies and the cookies are stored at client side. Alongwith a secret, when the cookie is passed by client in any request, the session variables are validated and the user session continues. When more requests are made by new clients, sessions are created and stored as cookies in their machine and when the subsequent requests are made along with cookie, after validating cookie data the user is authenticated. So no dependency on database is created and the time in validating data from database is saved. Even if the users delete the cookies from their local machine, in subsequent requests user is authenticated again and new cookies are created.