# CMPE273: Enterprise Distributed Systems Lab 2 Kafka And MongoDB

****

**By: Sunny Udhani**

**Student ID: 012457289**

**Repository:** <https://github.com/sunny-udhani/file_sharing_KAFKA_MONGO>

**For Github**

**Username – sunny-udhani**

**Pass – suns886633**

## **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

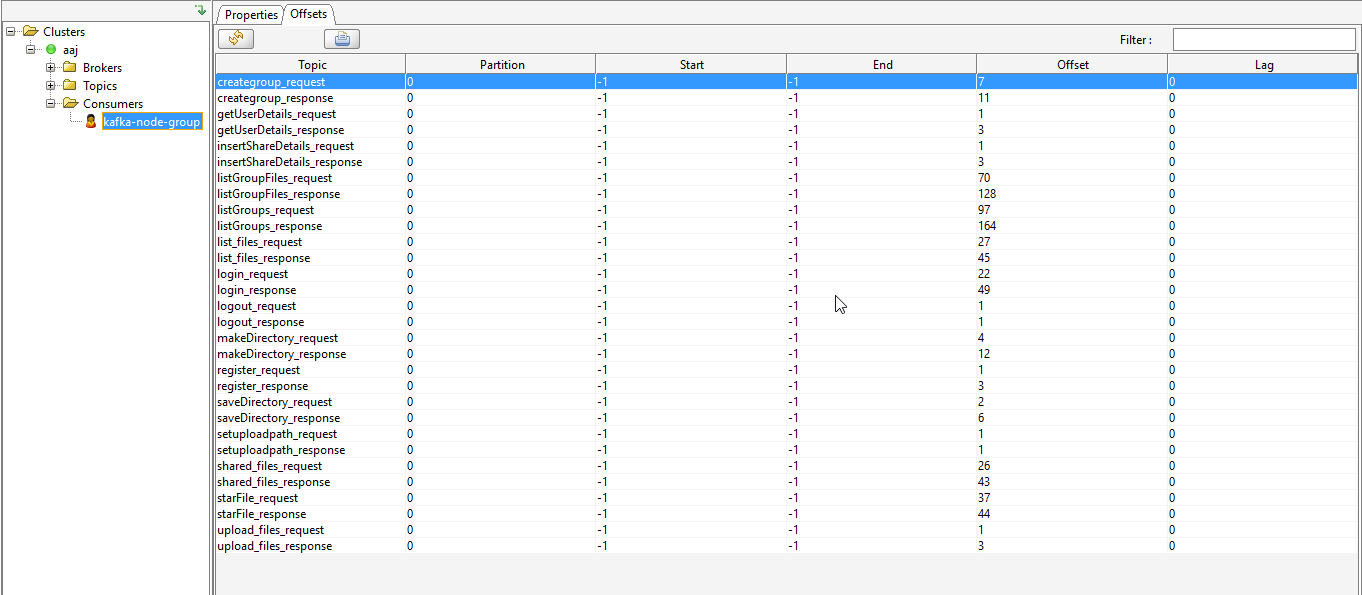
kafka-consumer

kafka-consumer

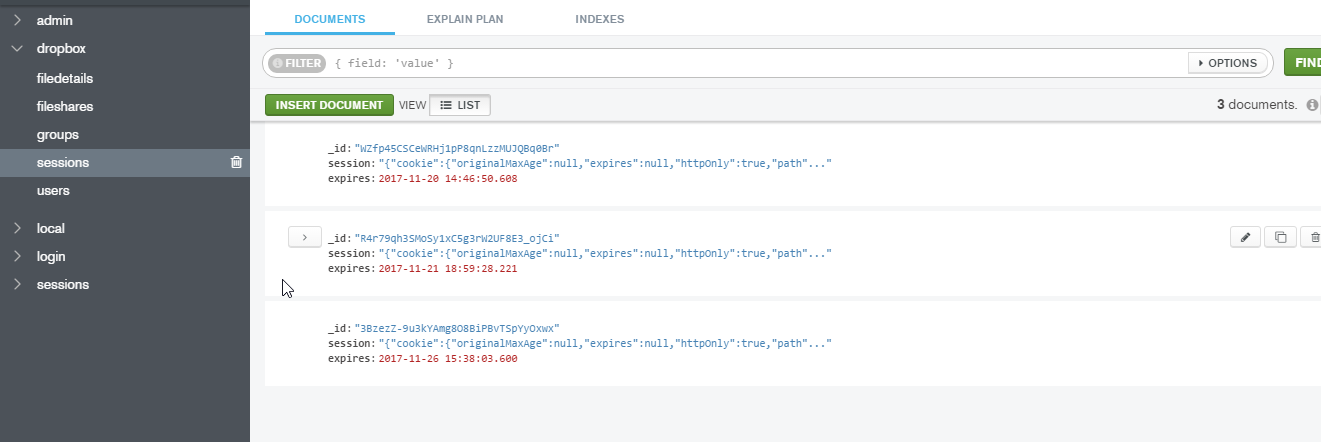
kafka-consumer

MongoDB

### Kafka-Topics



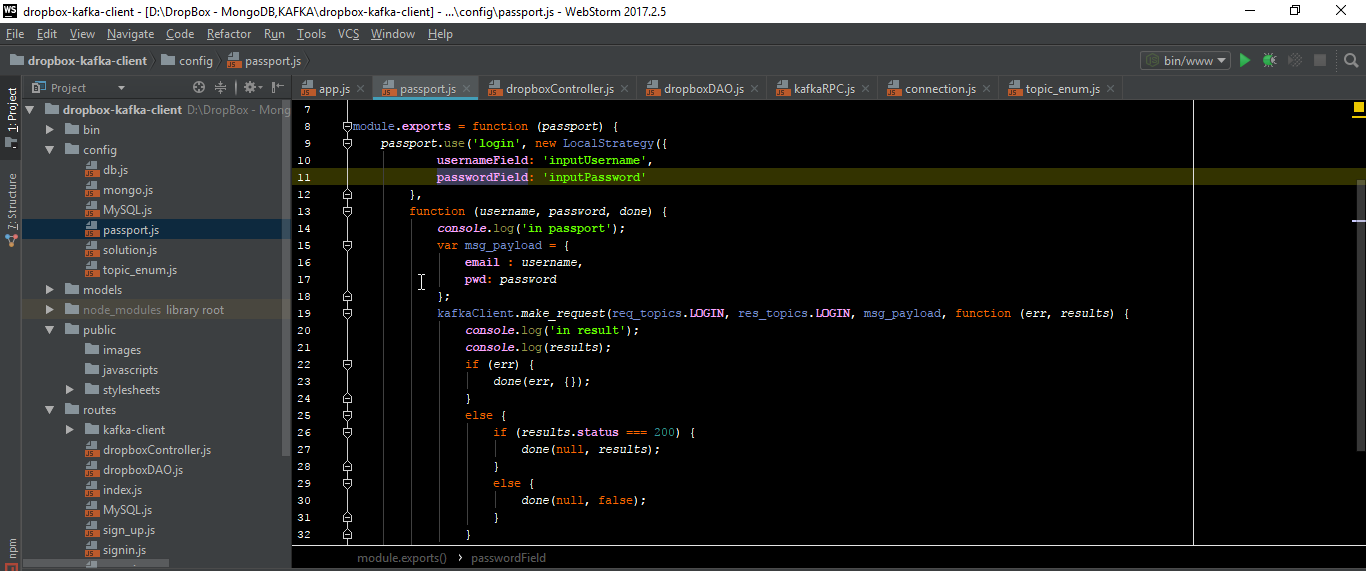
### MongoDB for Storing sessions



**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());

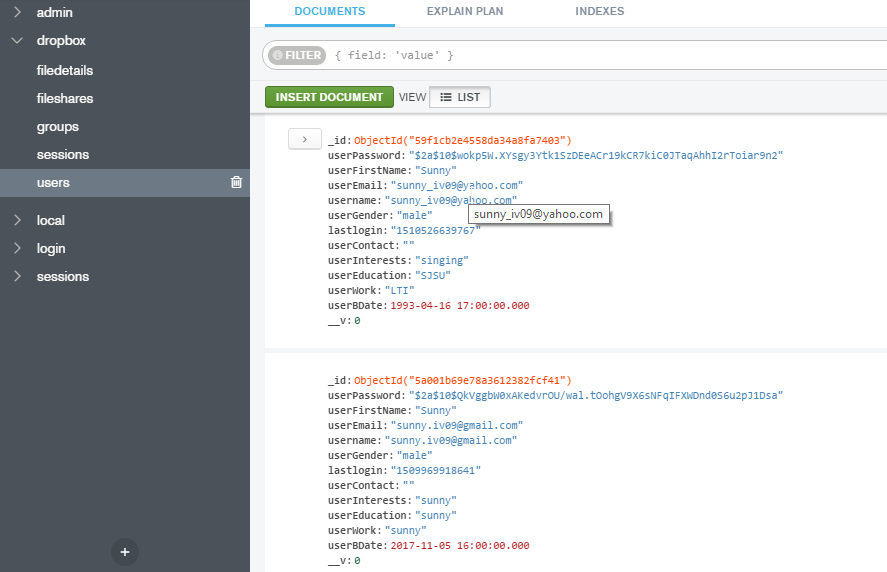
### PassportJS for Authentication



**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**

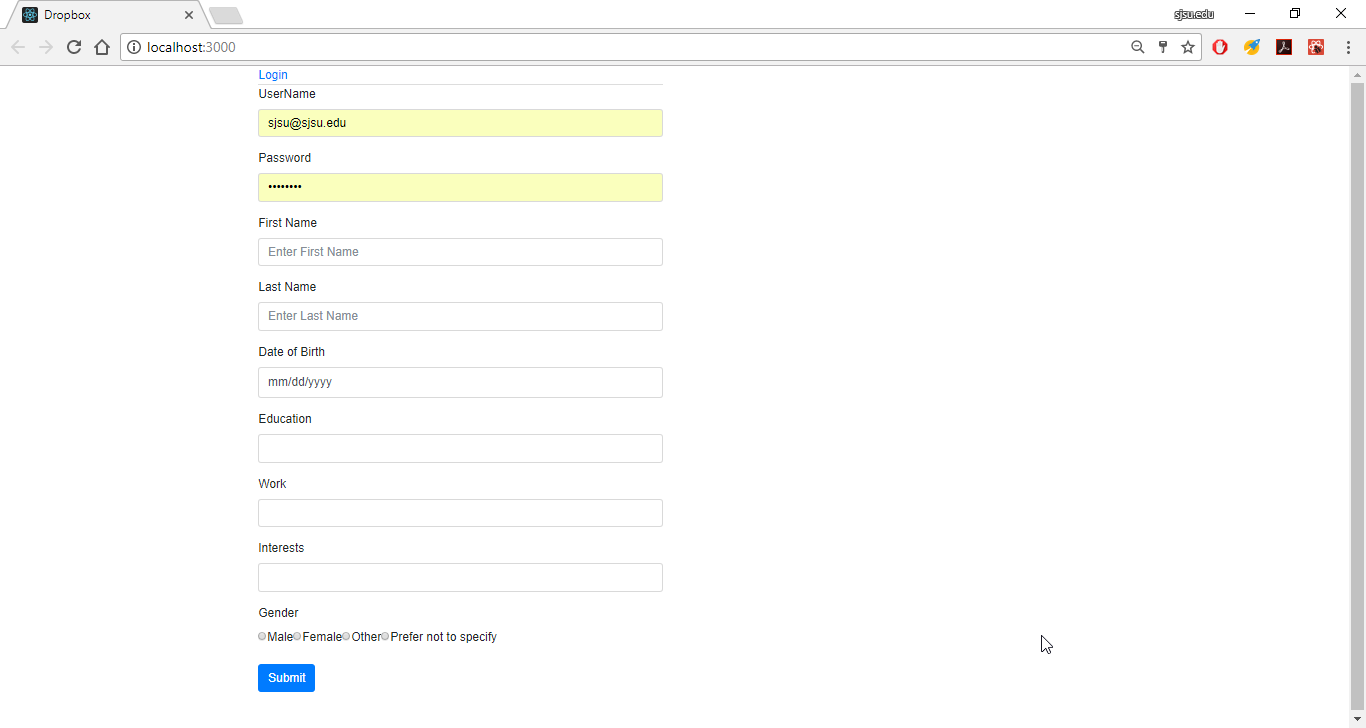
var poolSize = 10;  
var queueSize = 10;  
  
var pool = [];  
var queue = [];  
var queueCount = 0;  
  
var queueNotifier = new **Map**();  
  
  
function *CreateConnectionPool*() {  
  
 for (var i = 0; i < poolSize; i++) {  
 var connection = ***mongoose***.connect(dbURI);  
 pool.push(connection);  
 }  
 ***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:**

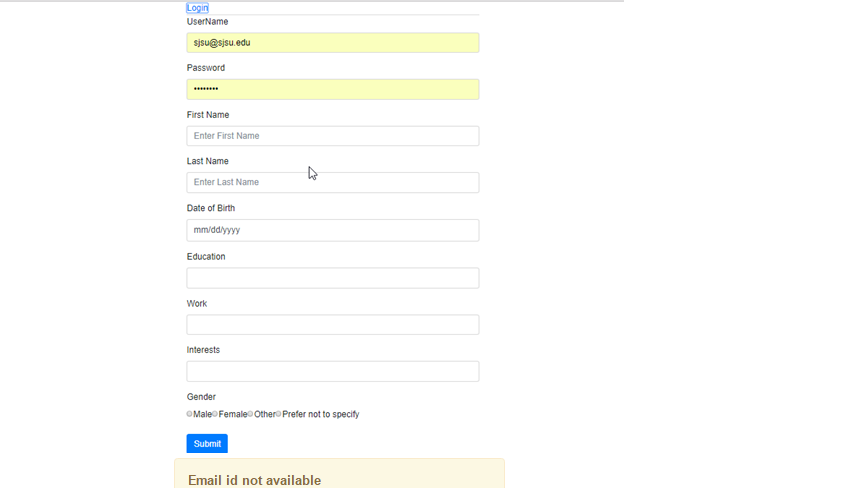
1. **Signup page**: Allows the user to sign up for the application so that they can use it to share and upload their files. It takes a variety of user input to provide a personalized experience on login.
2. **Login page**: Allows the user to login to the application and create user session. The user 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:**

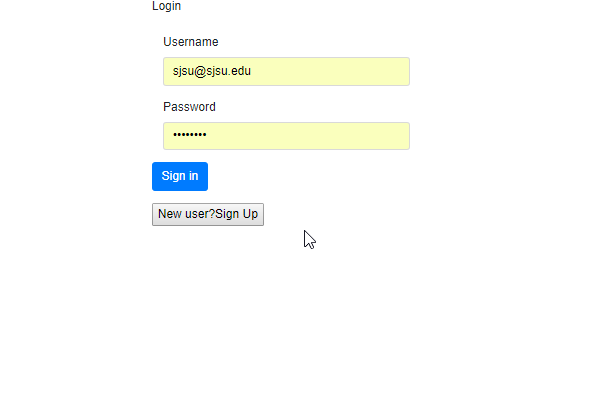
**Screen captures of dropbox prototype:**

**Register**: 

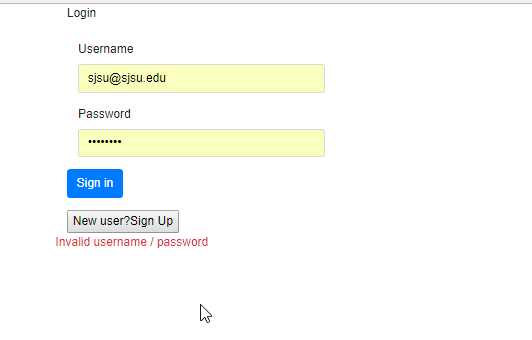
**Validation**: If user tries to register with already existing email-address, it will display message as show below:



**Login:**

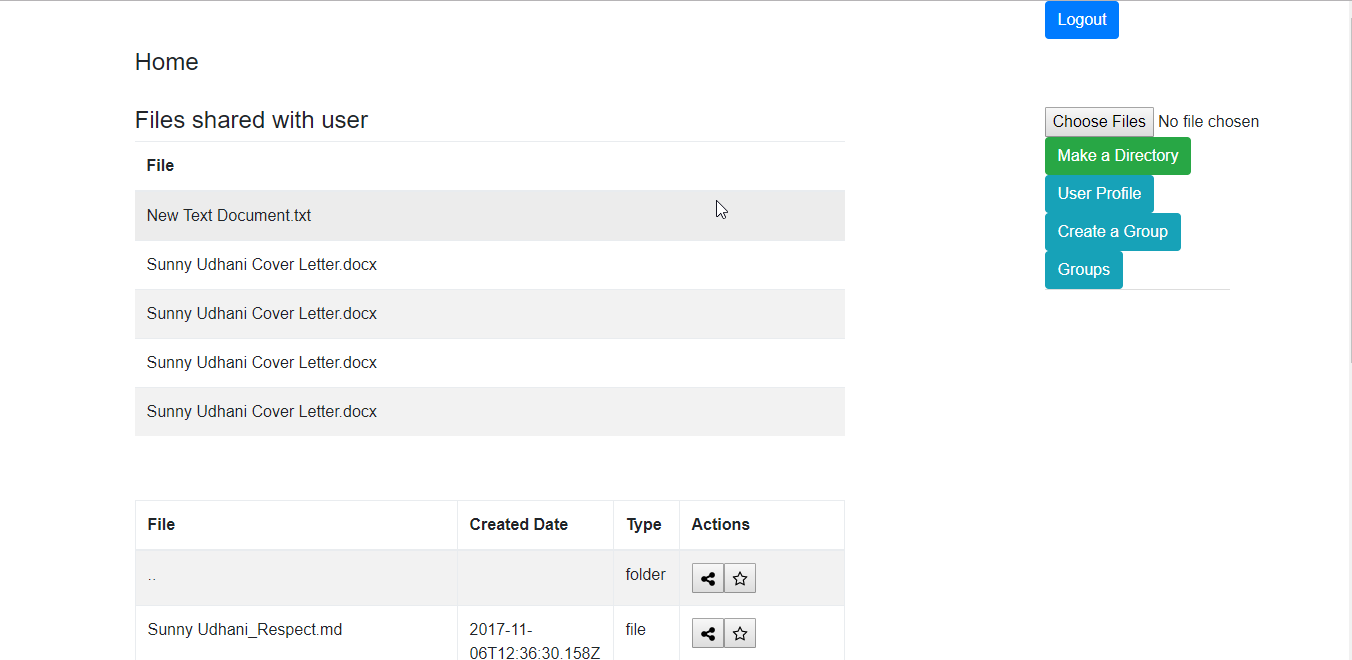
****

**Validation: If incorrect username or password**

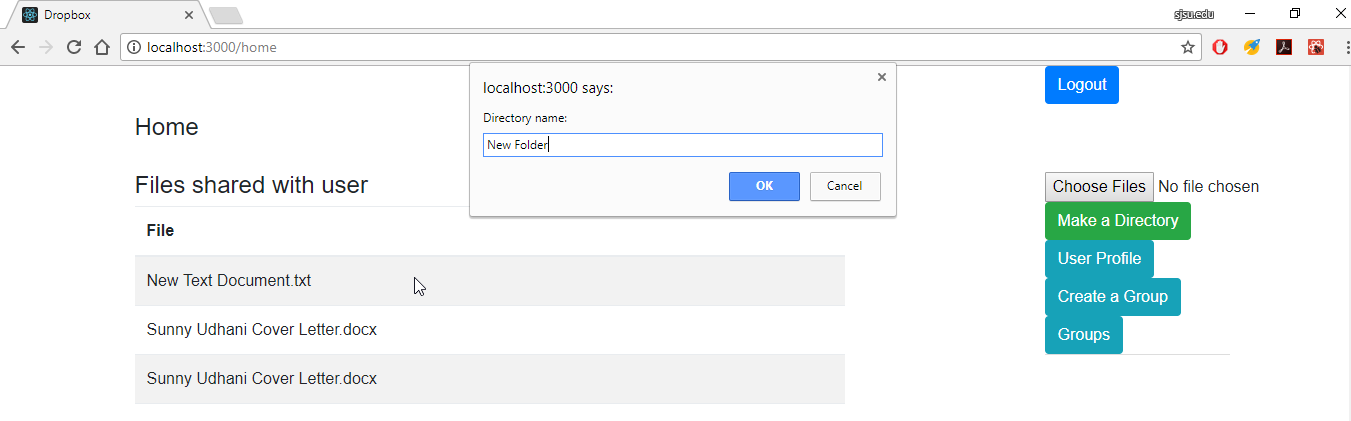
****

**Home:**

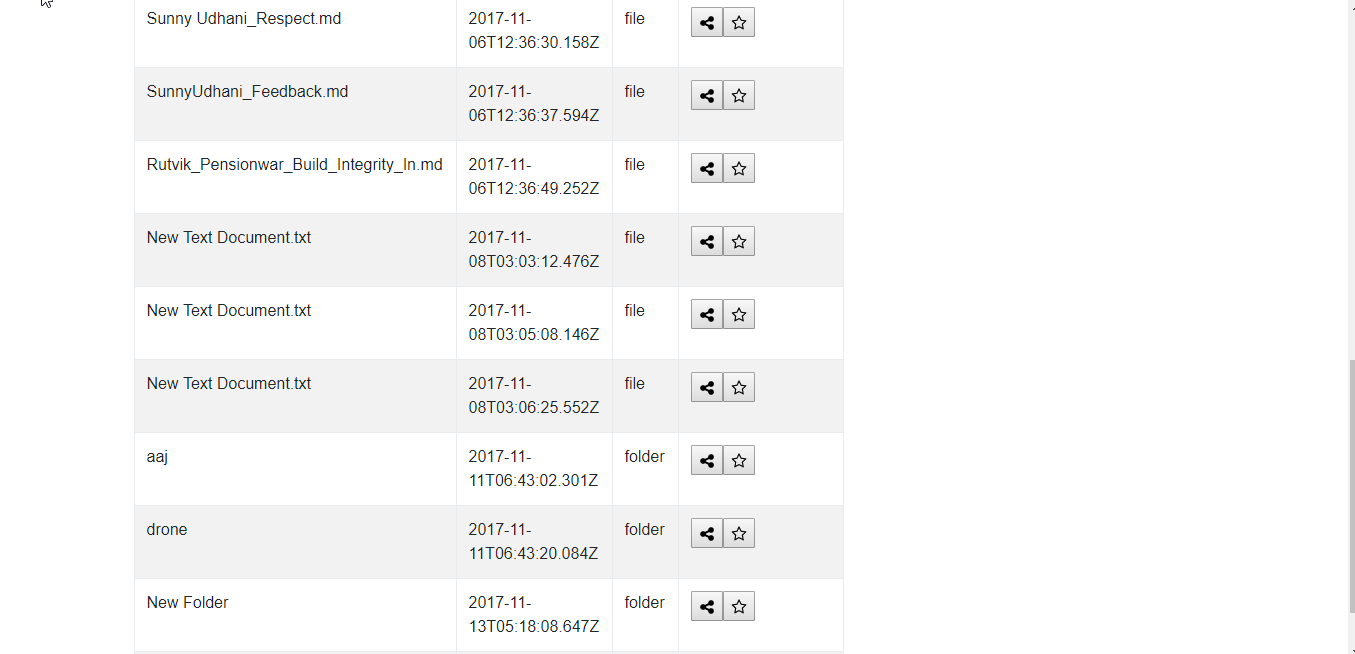
User gets to the home page after successful login. The home page shows a list of files that are uploaded and starred by user. It also has options to logout of system, create a directory or to view user profile.

******

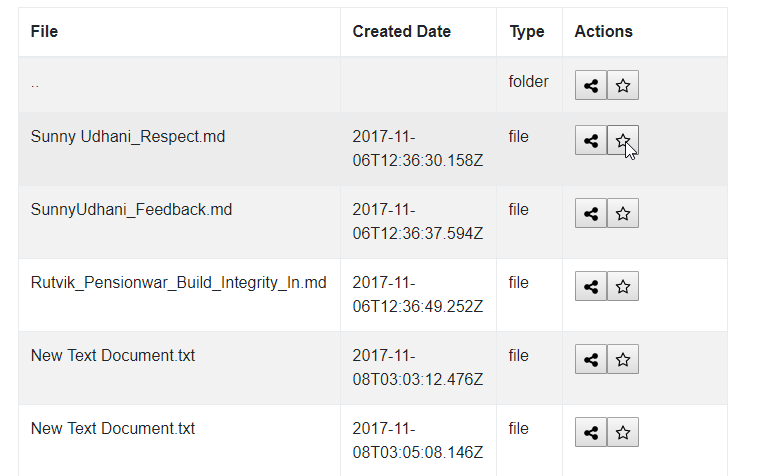
**Create a directory:**

****

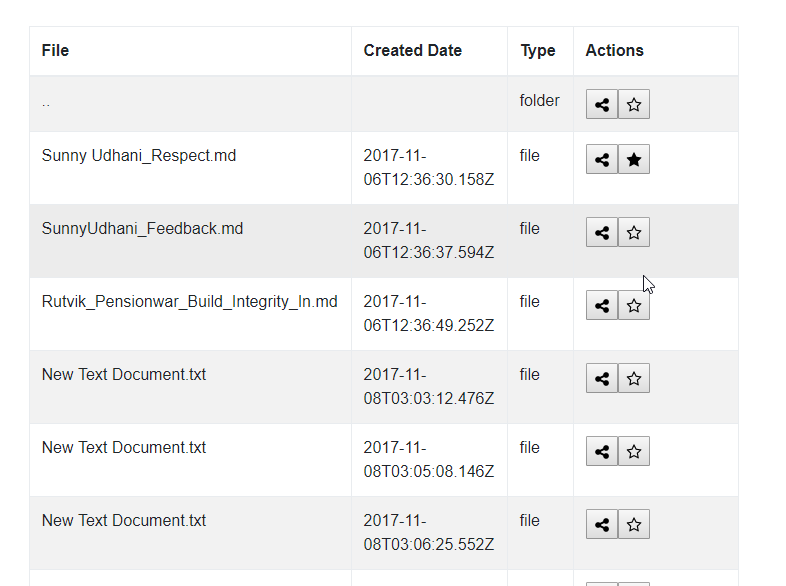
**Result:**

******

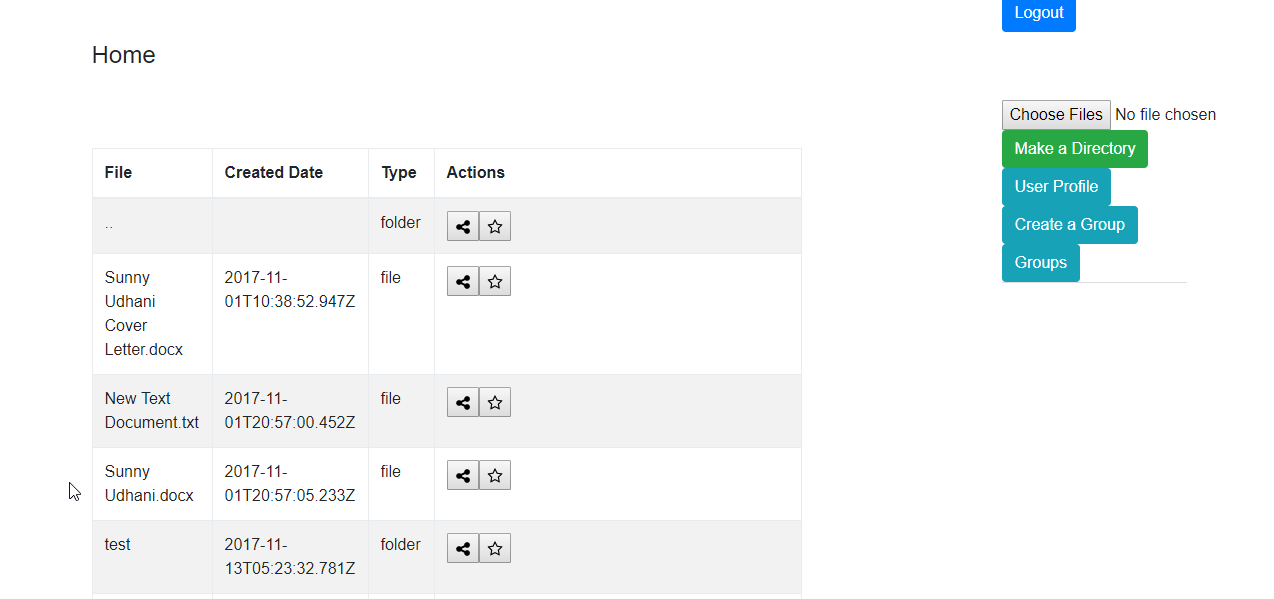
**Star a directory:**

****

**Result:**

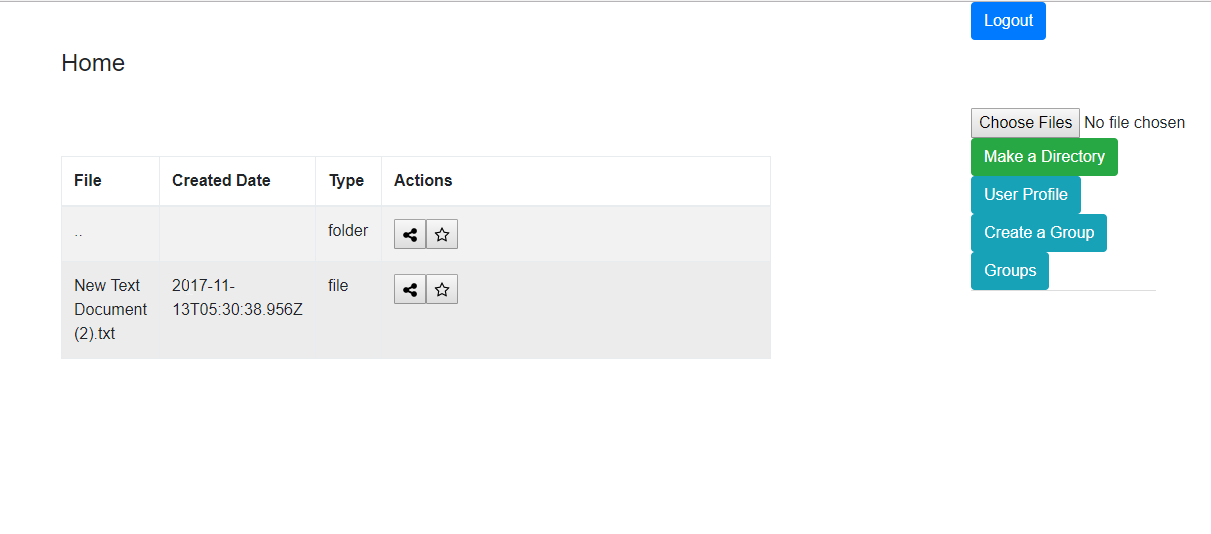
****

**Go inside Directory:**

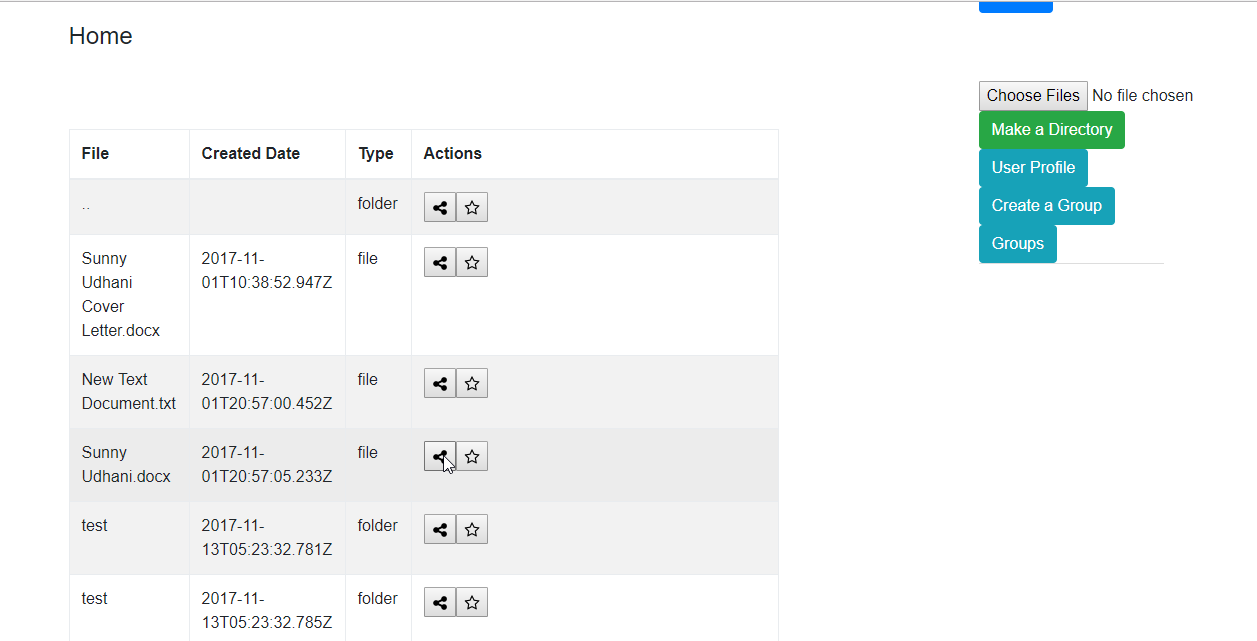
****

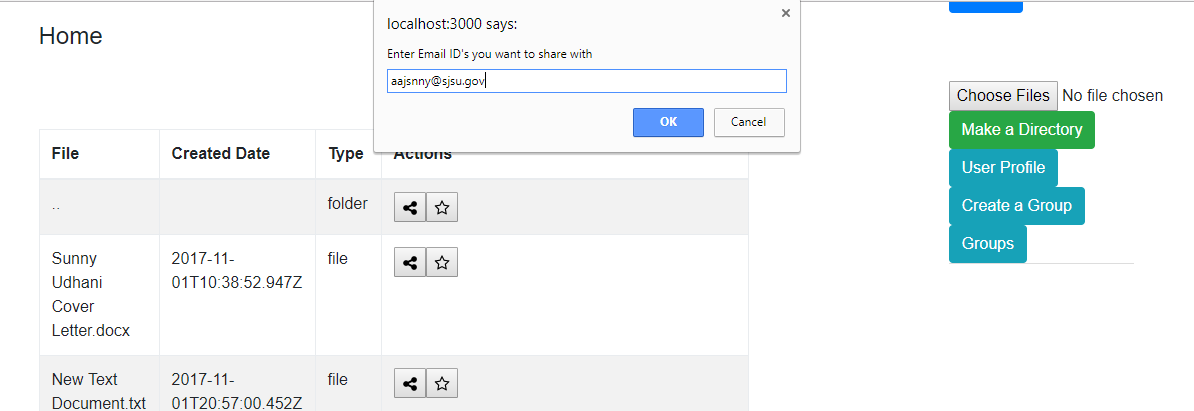
**Result:**

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

****

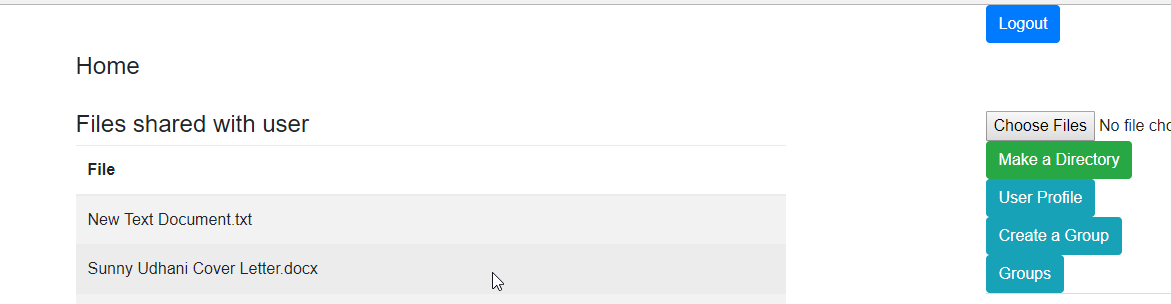
**Share file/directory**

****

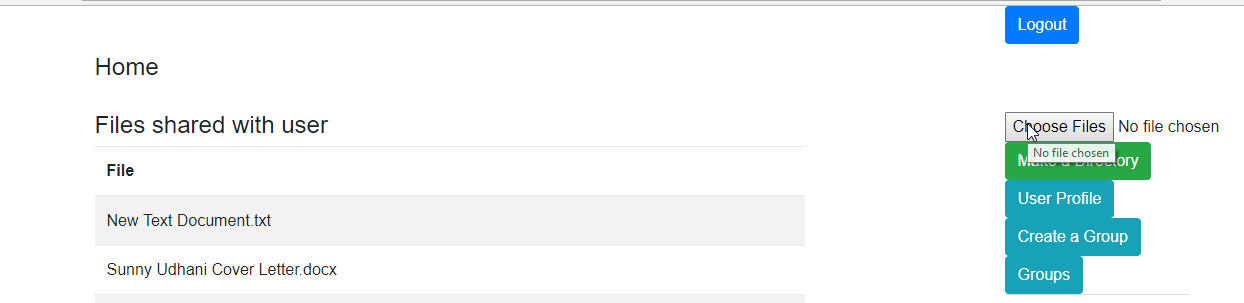
****

**Result:**

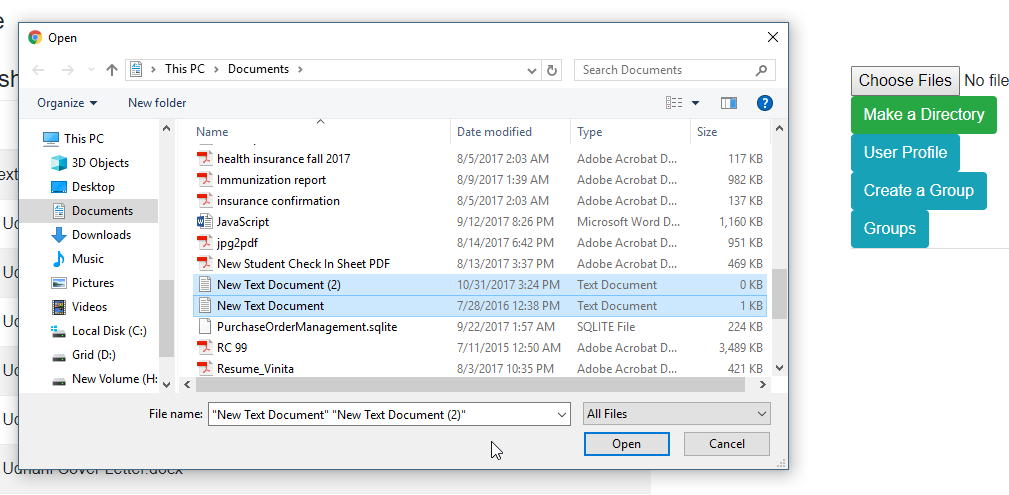
When I login with [aajsnny@gov.in](mailto:aajsnny@gov.in), I will be shown another table with files shared with the user.

****

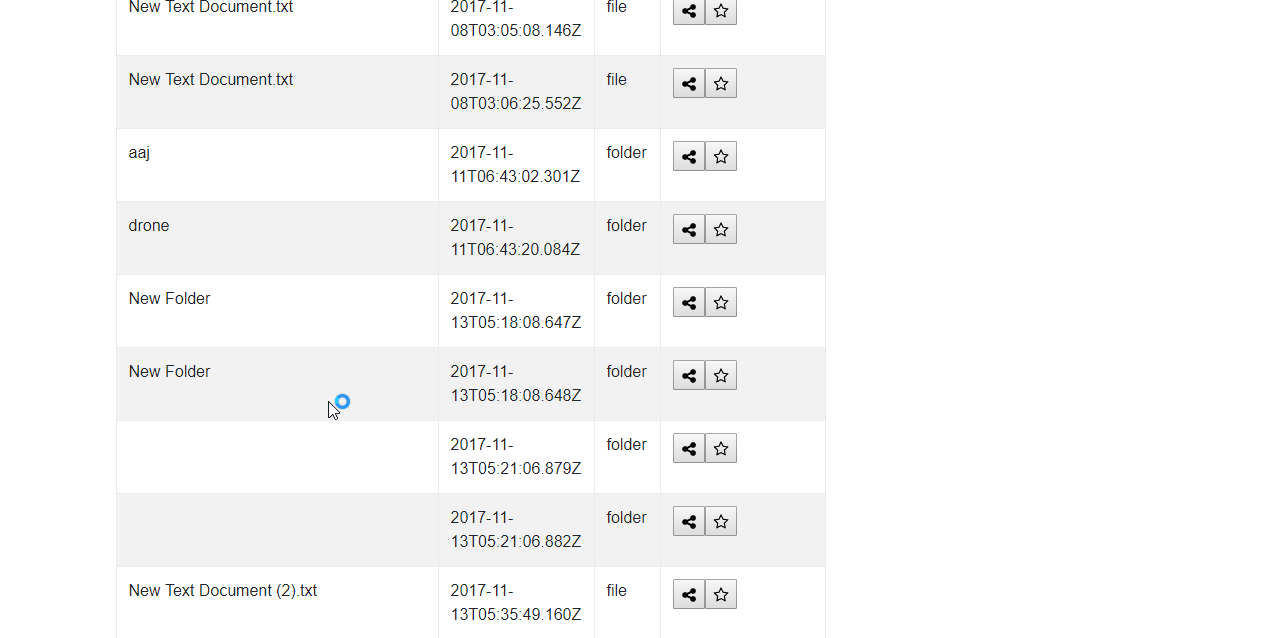
**Upload files:**

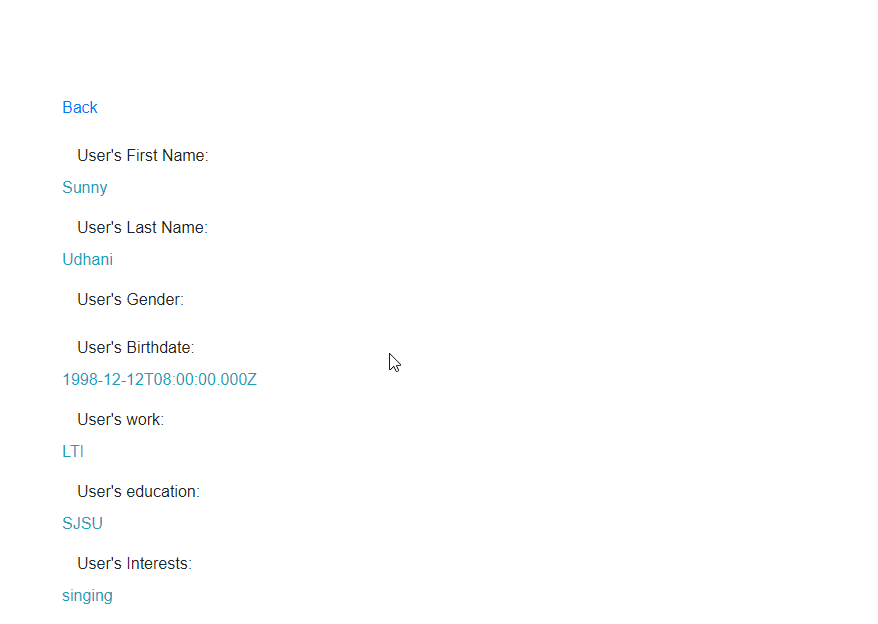
****

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

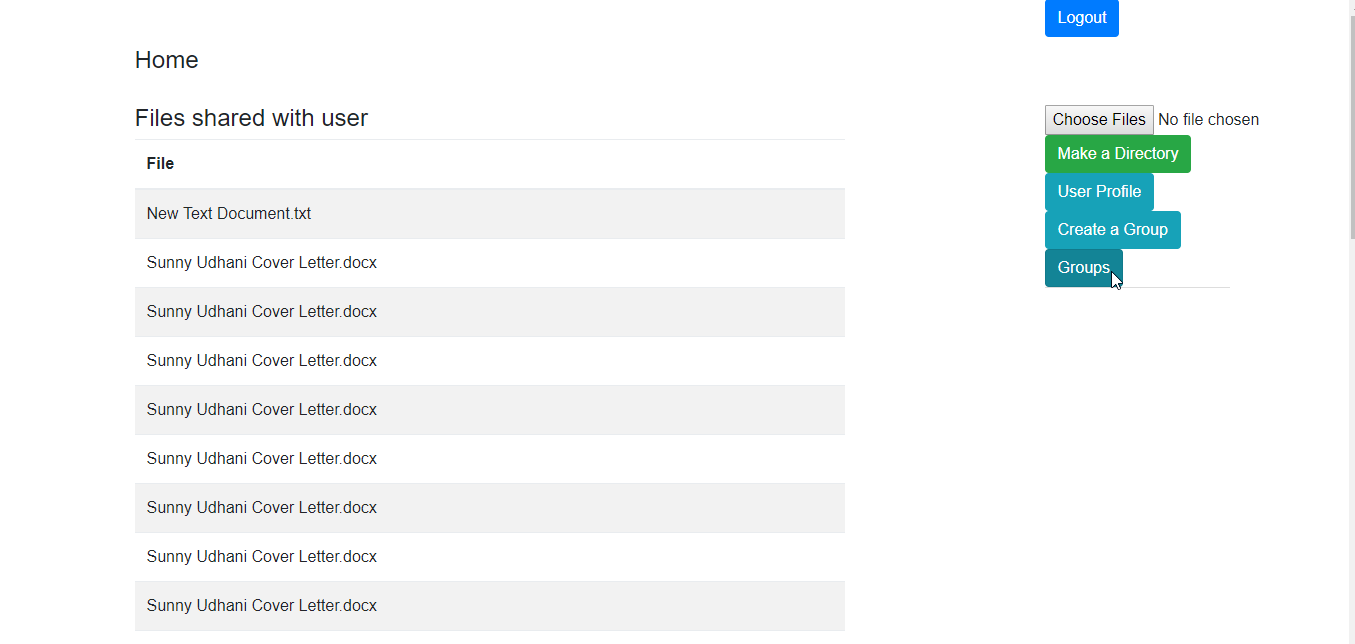


**Result:**

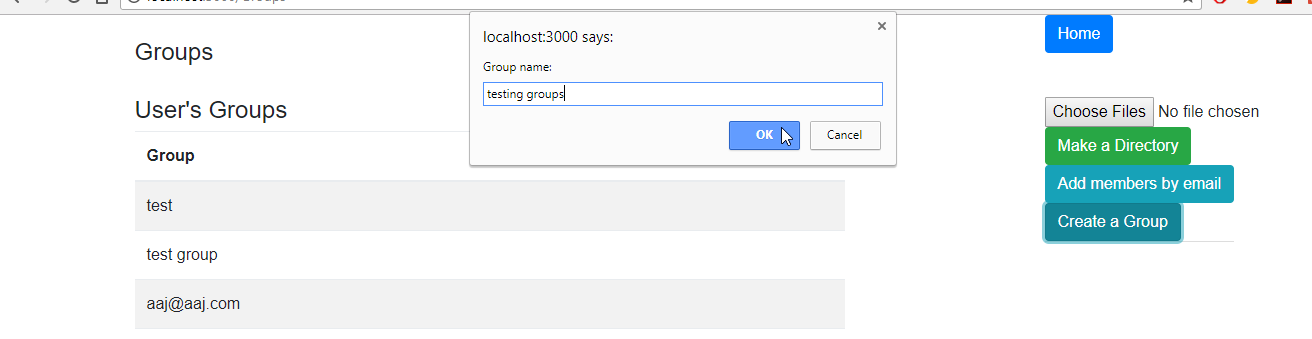
****

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

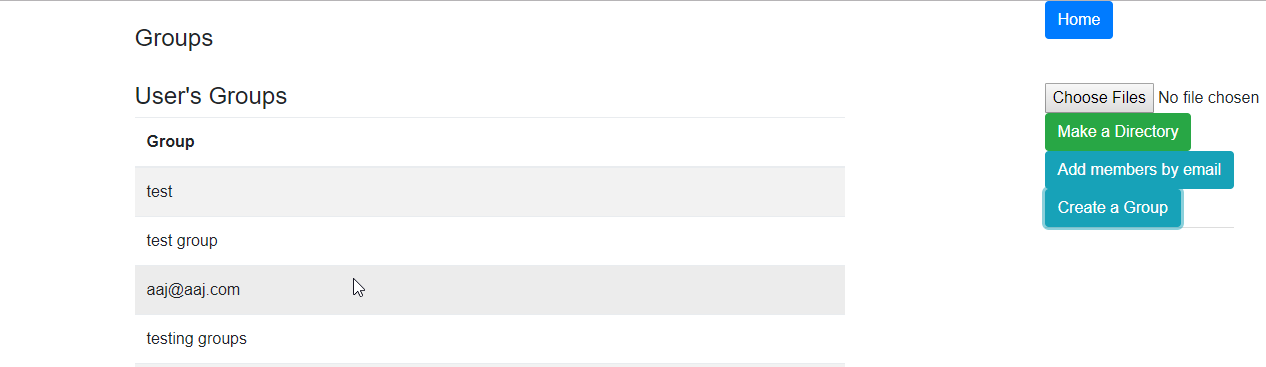
**Groups:**

****

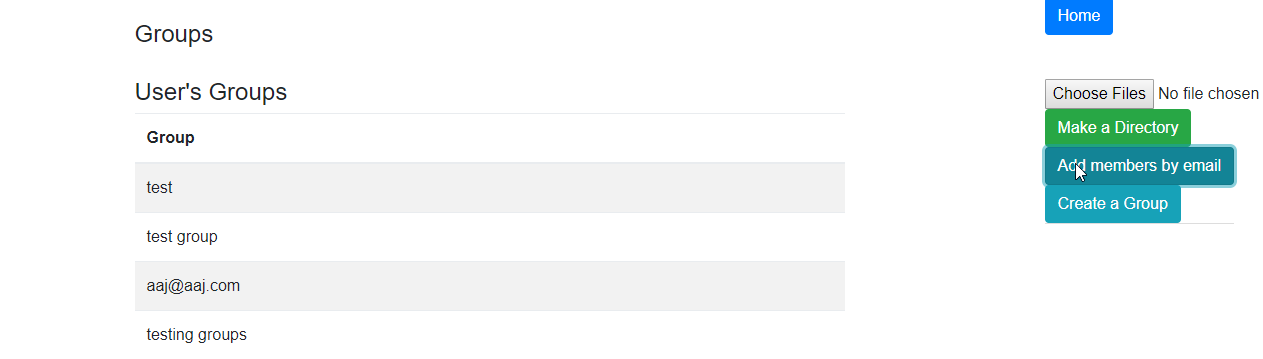
**Create Group:** allow user to create a group

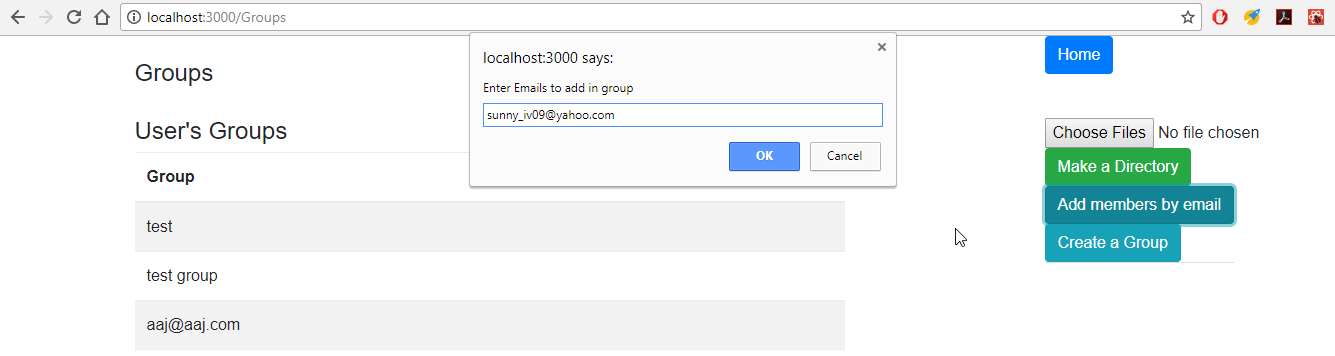
****

**Result:**

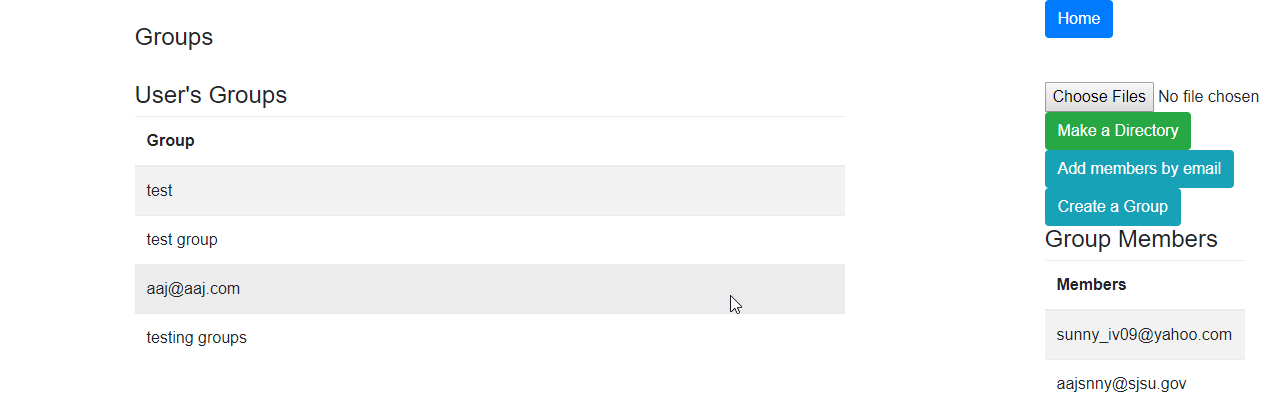
****

**Add members to Group:**

****

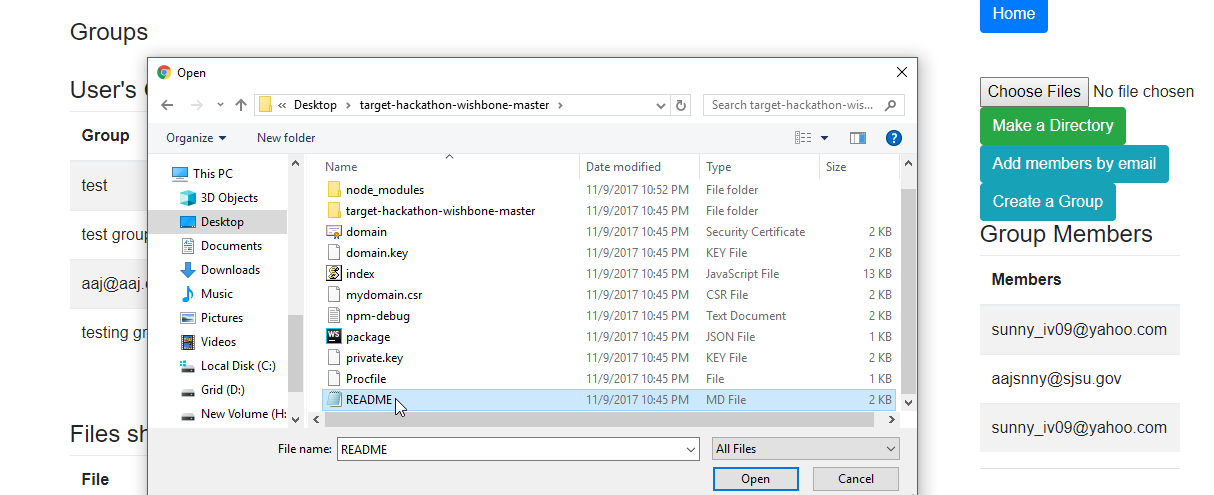
****

**Result:**

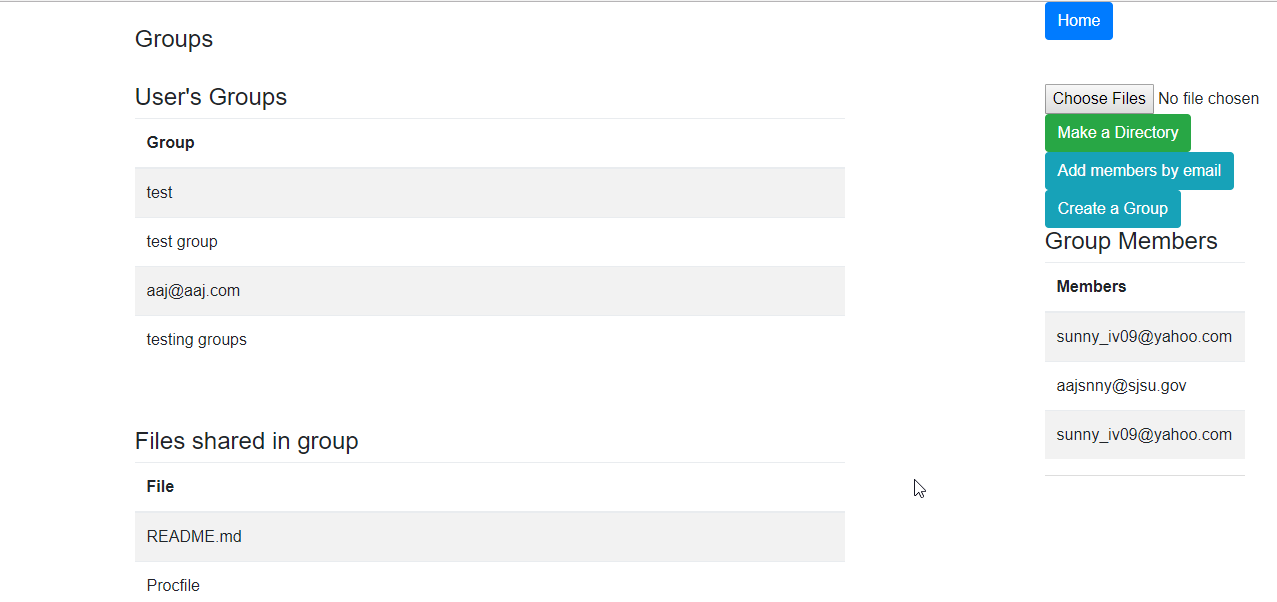
****

**Upload files to group:**

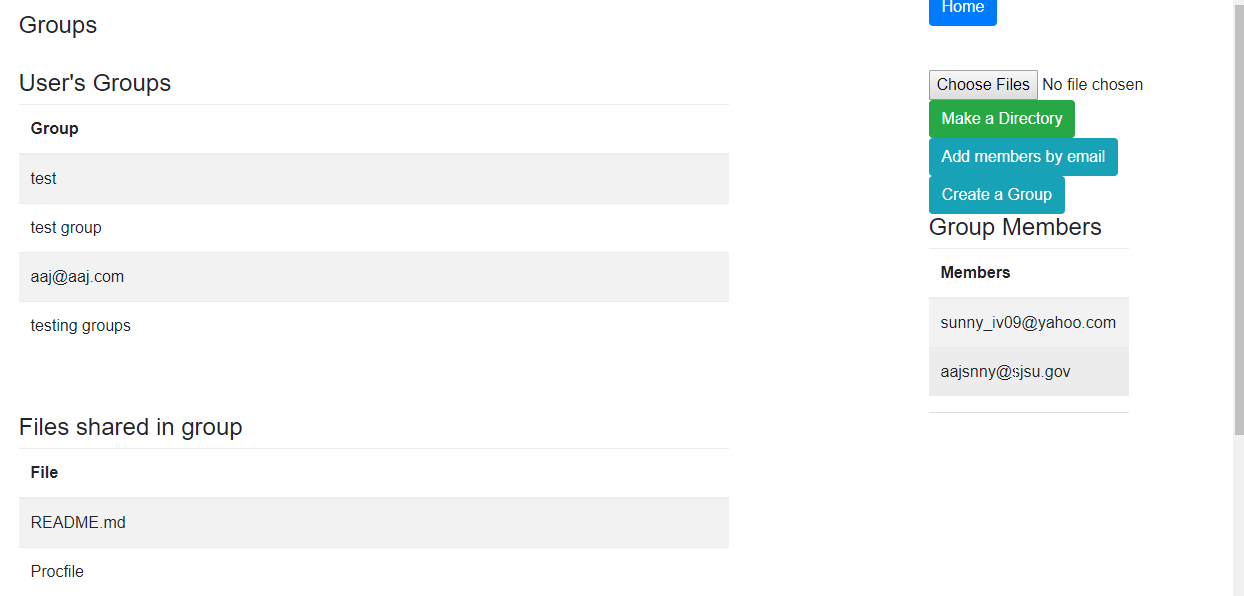
****

****

**Result:**

****

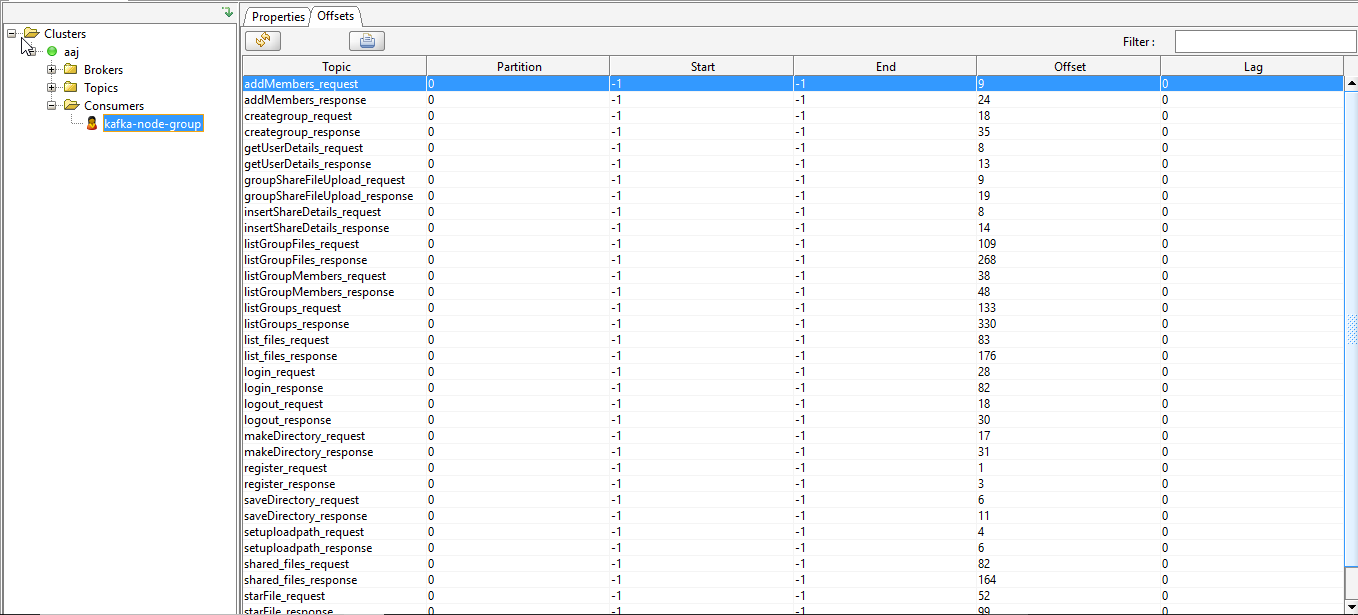
**Group member list:**

****

## **Performance**

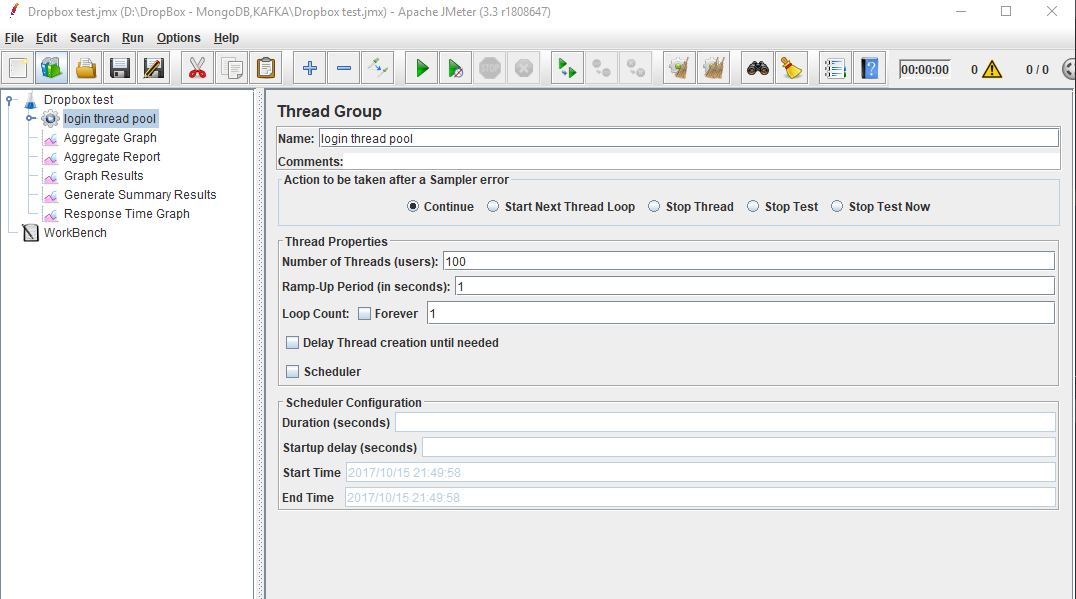
**JMETER TESTING**

**Offset counts for topics**

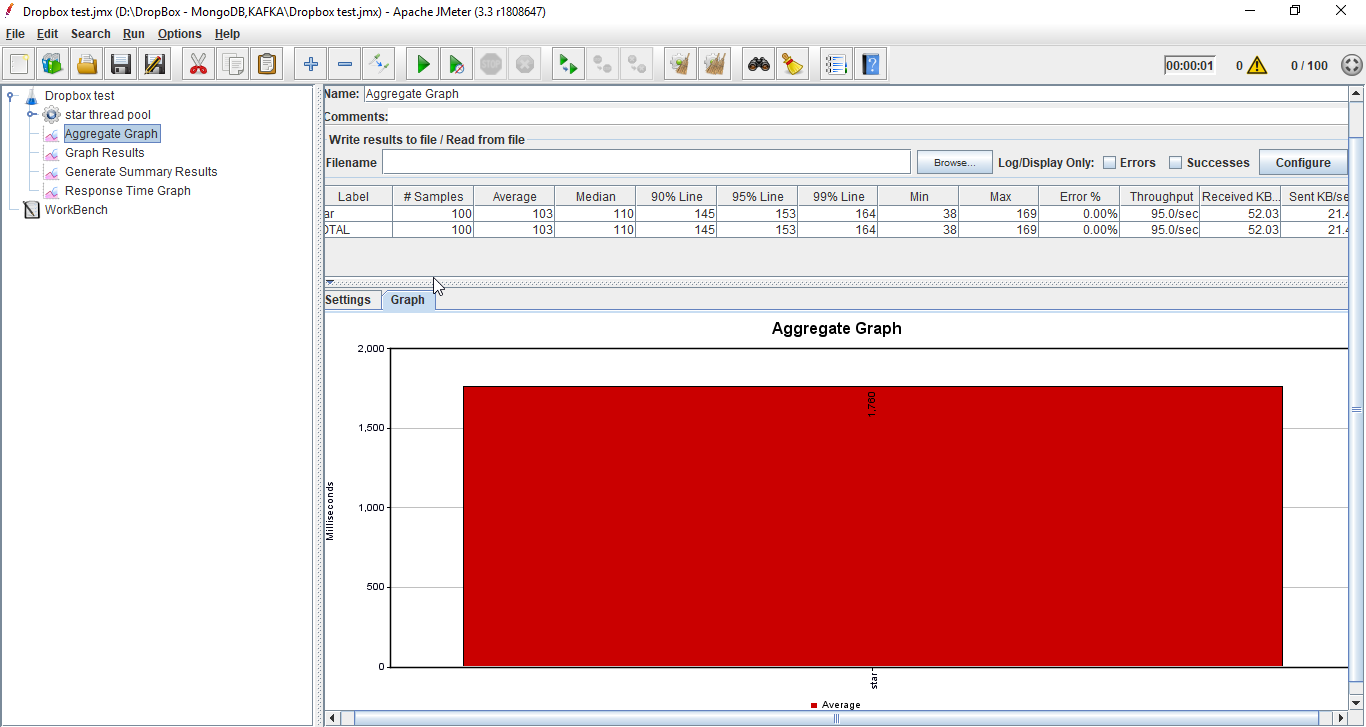
****

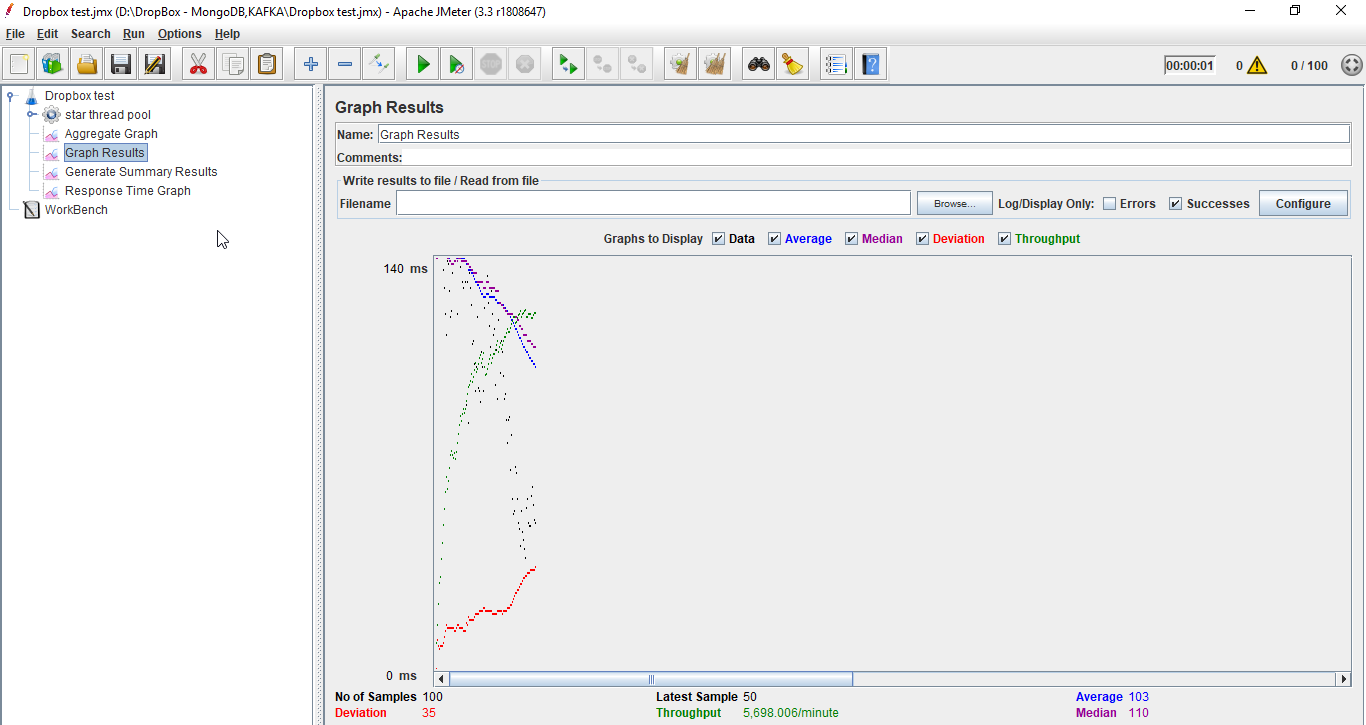
1. **For 100 concurrent users.**

**Setup**-

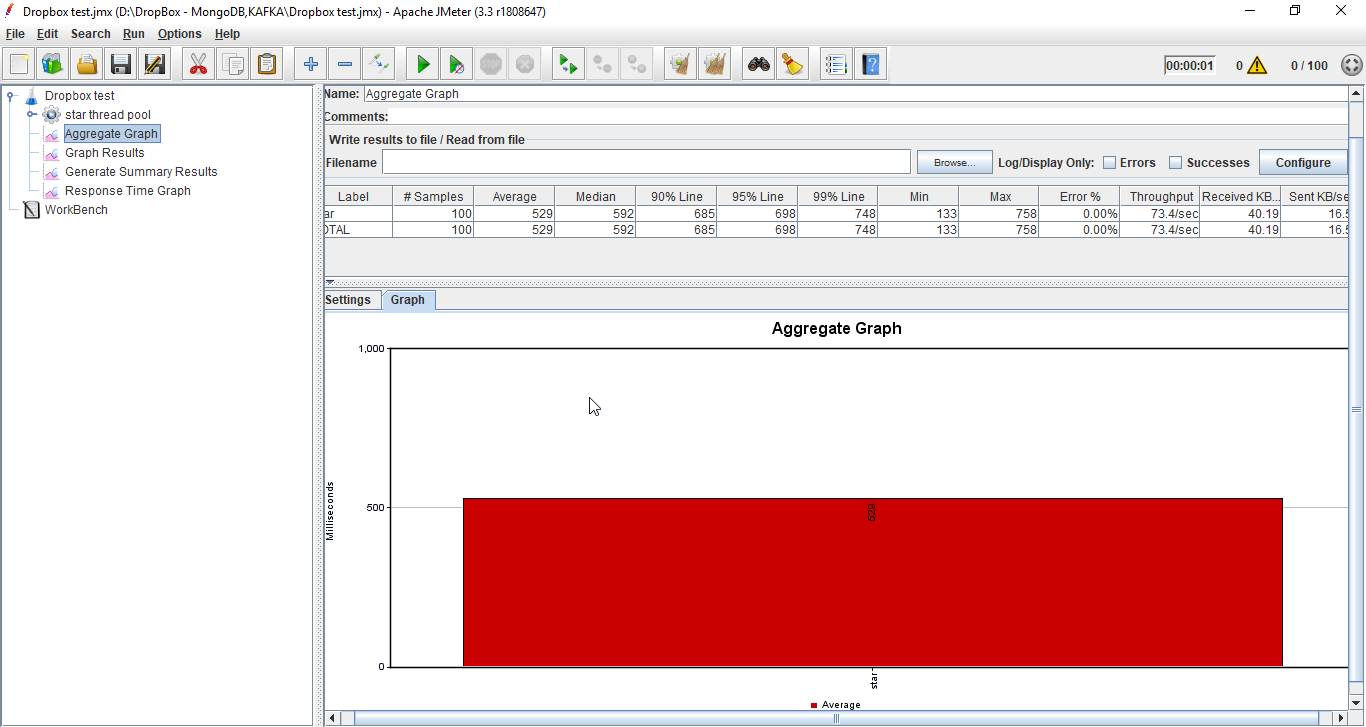


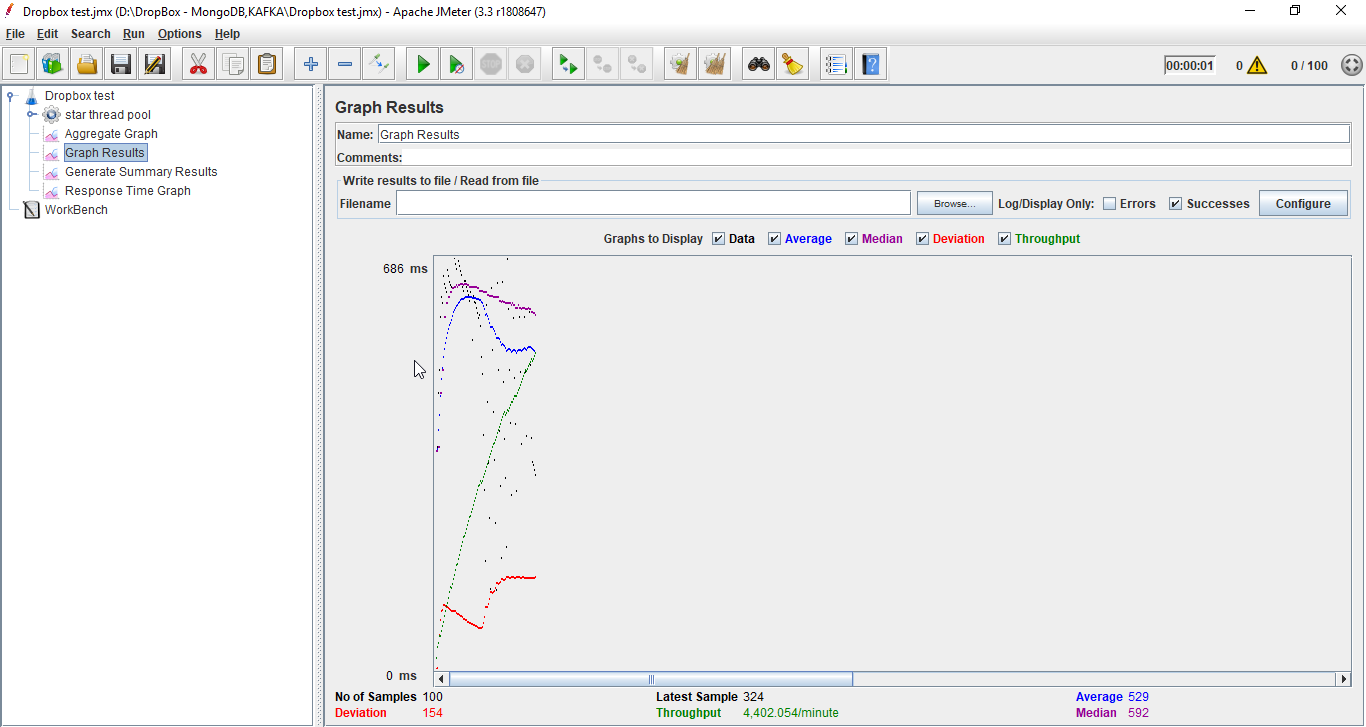
**Without Connection Pooling**:



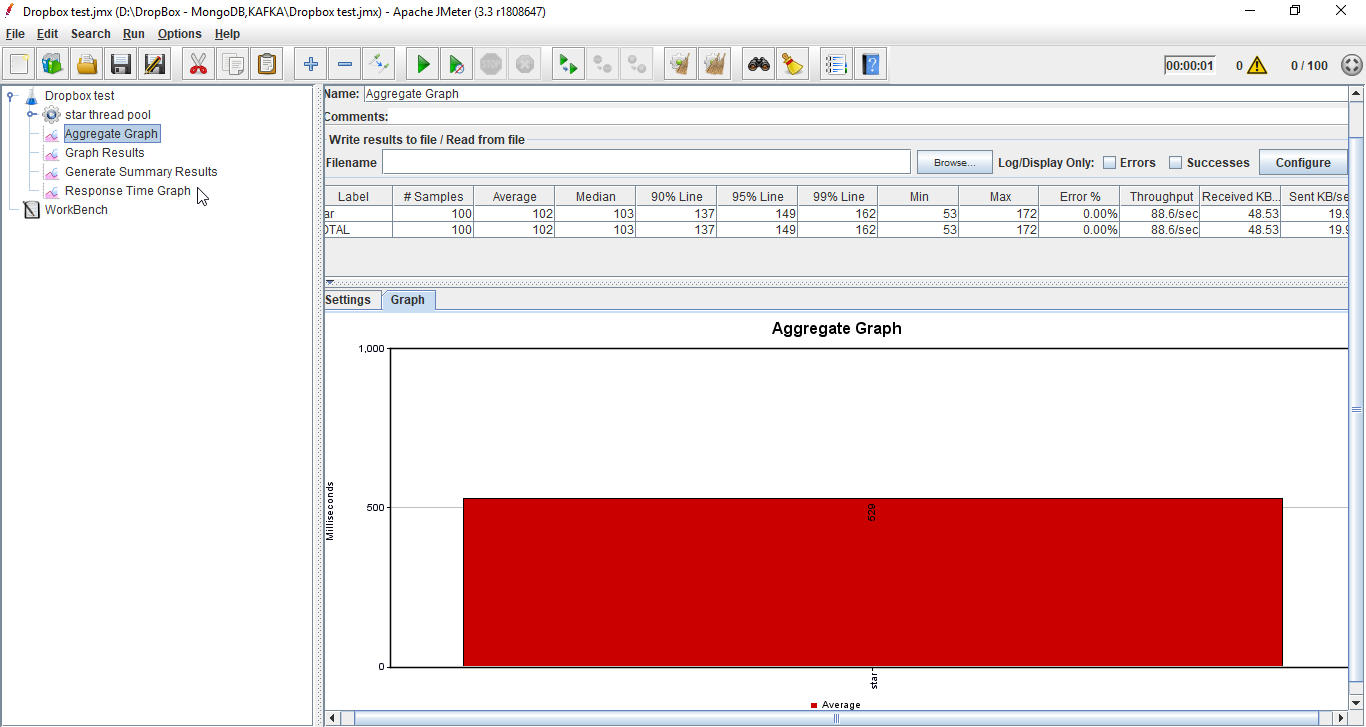


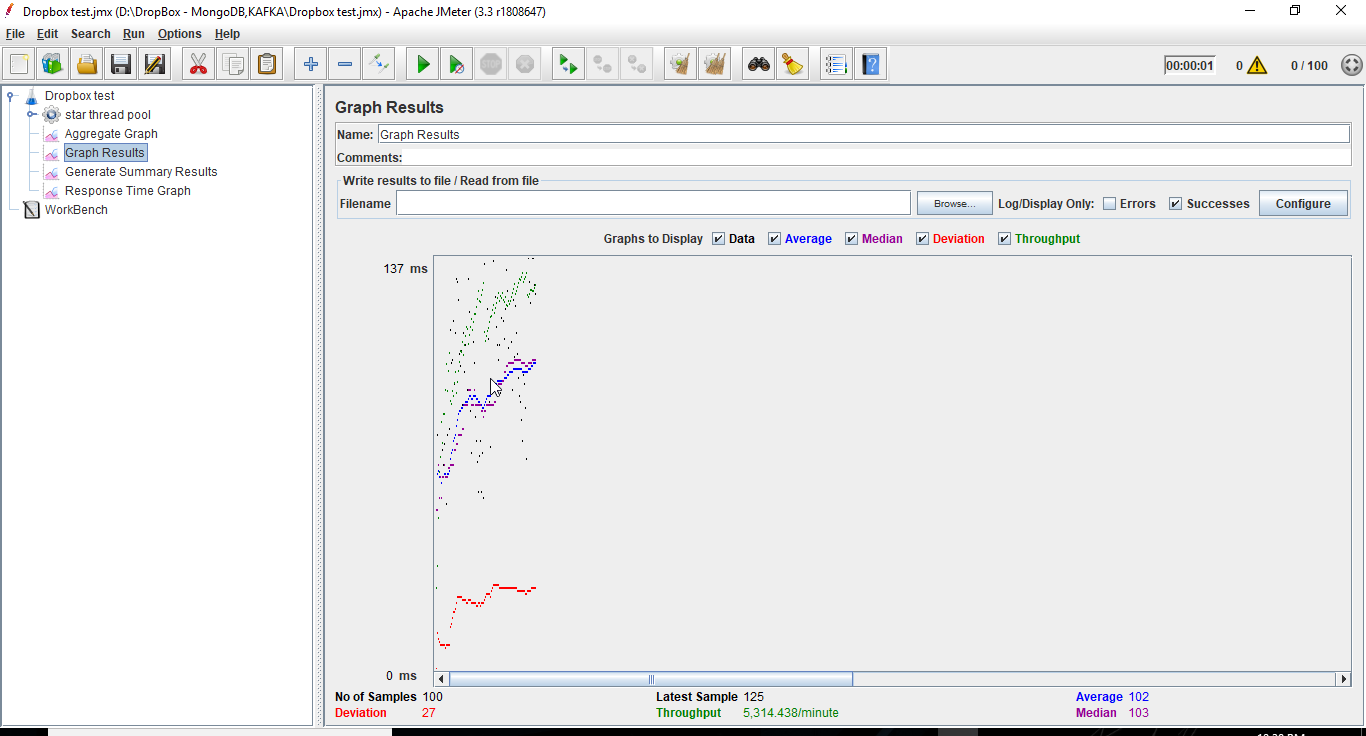
**With Connection Pooling**: DB provided



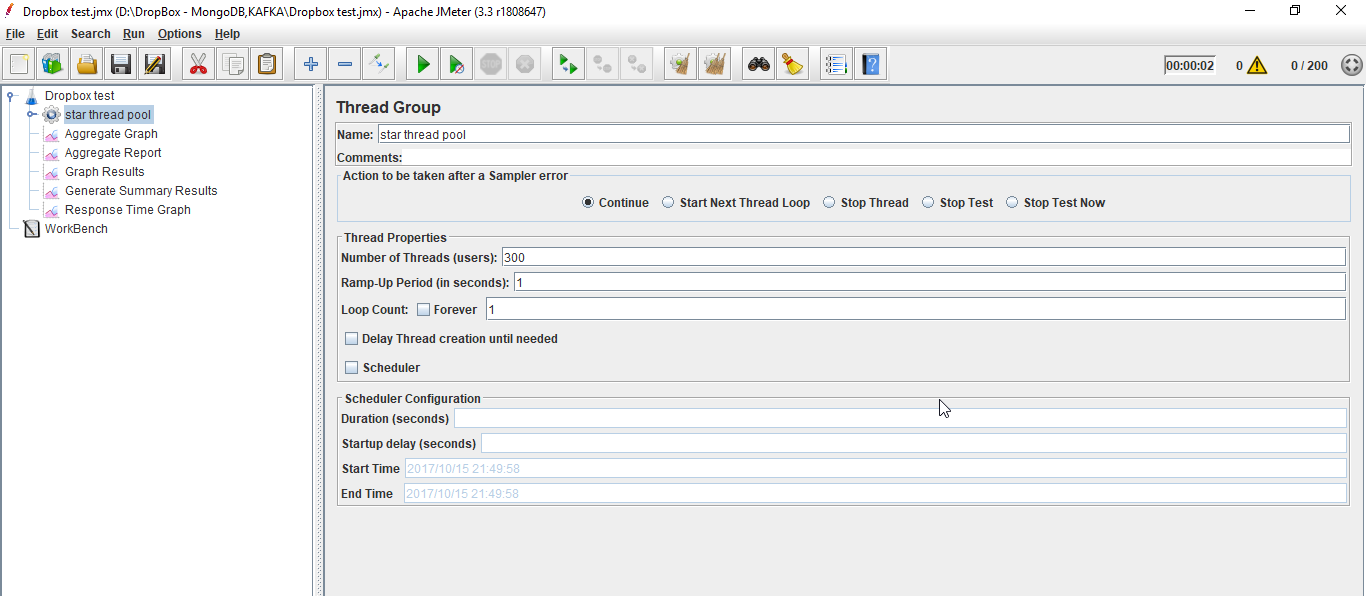
****

**With Connection Pooling**: own connection pooling implementation

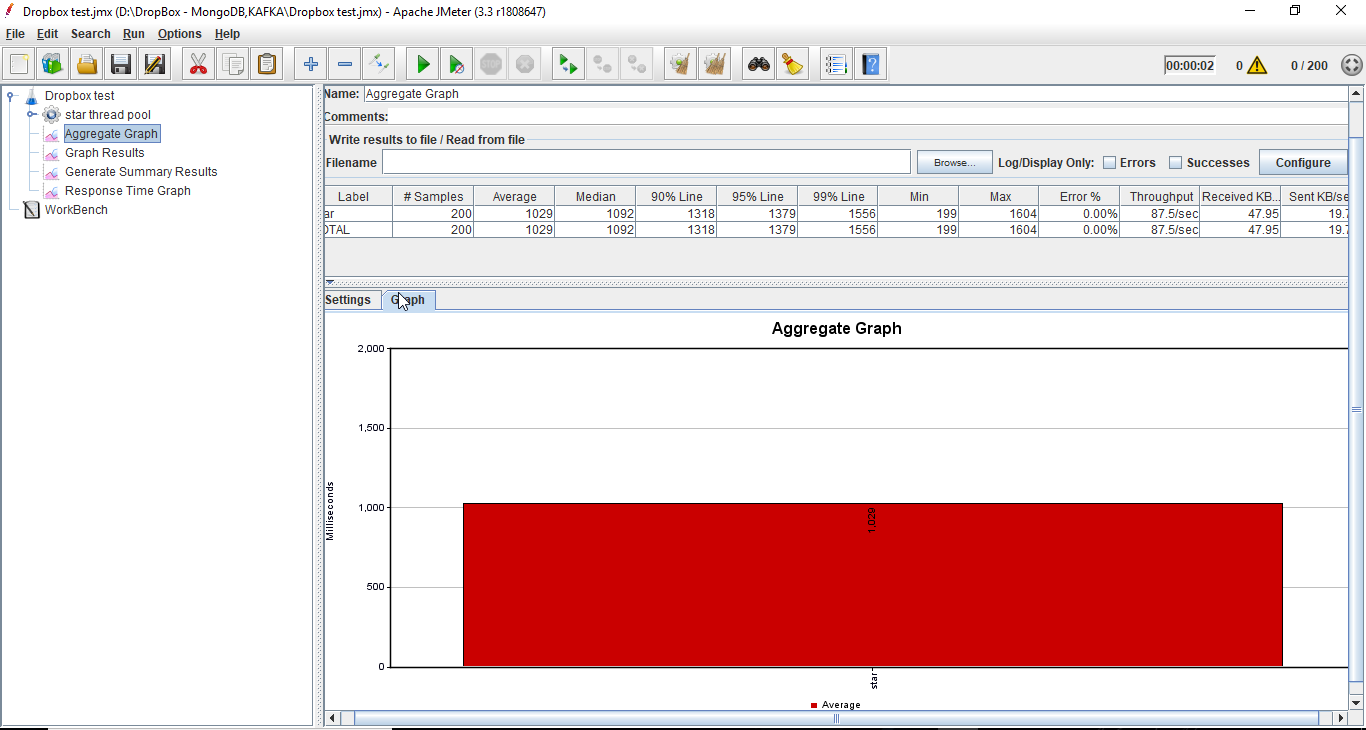


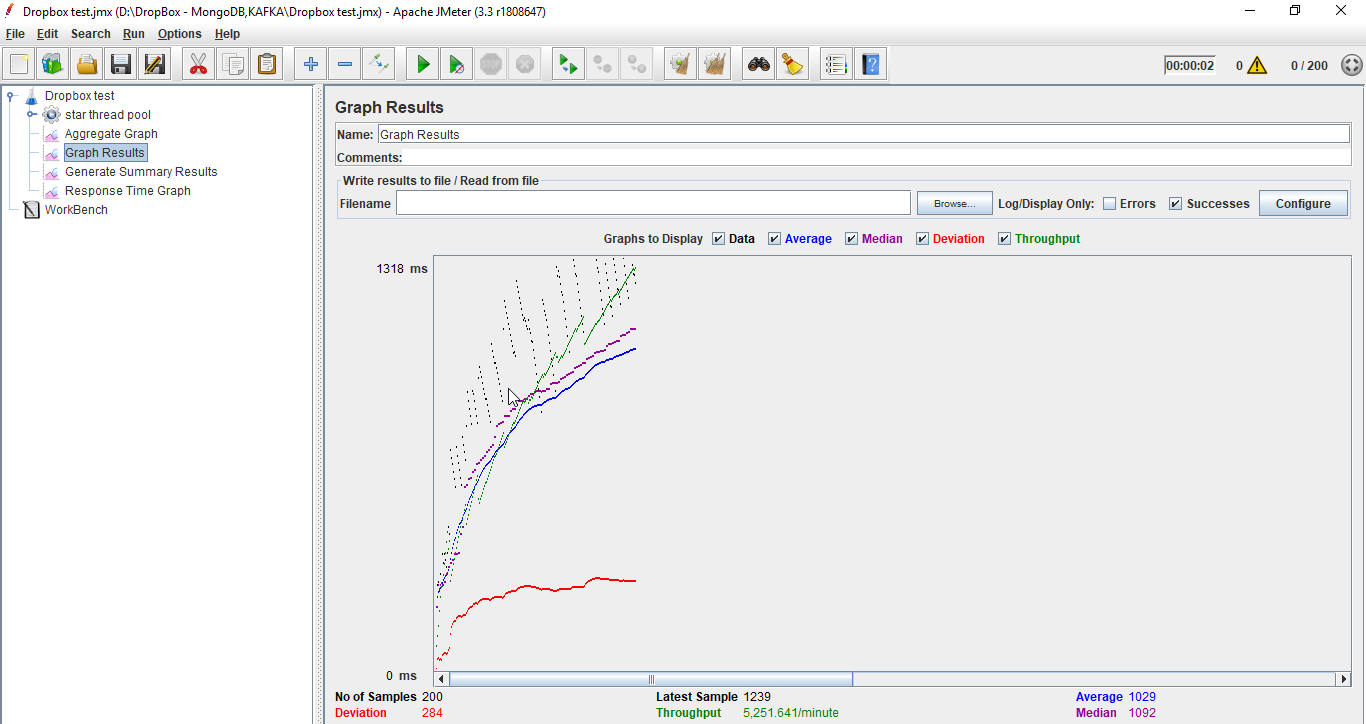


1. For 200 concurrent users:  
   **Setup**-

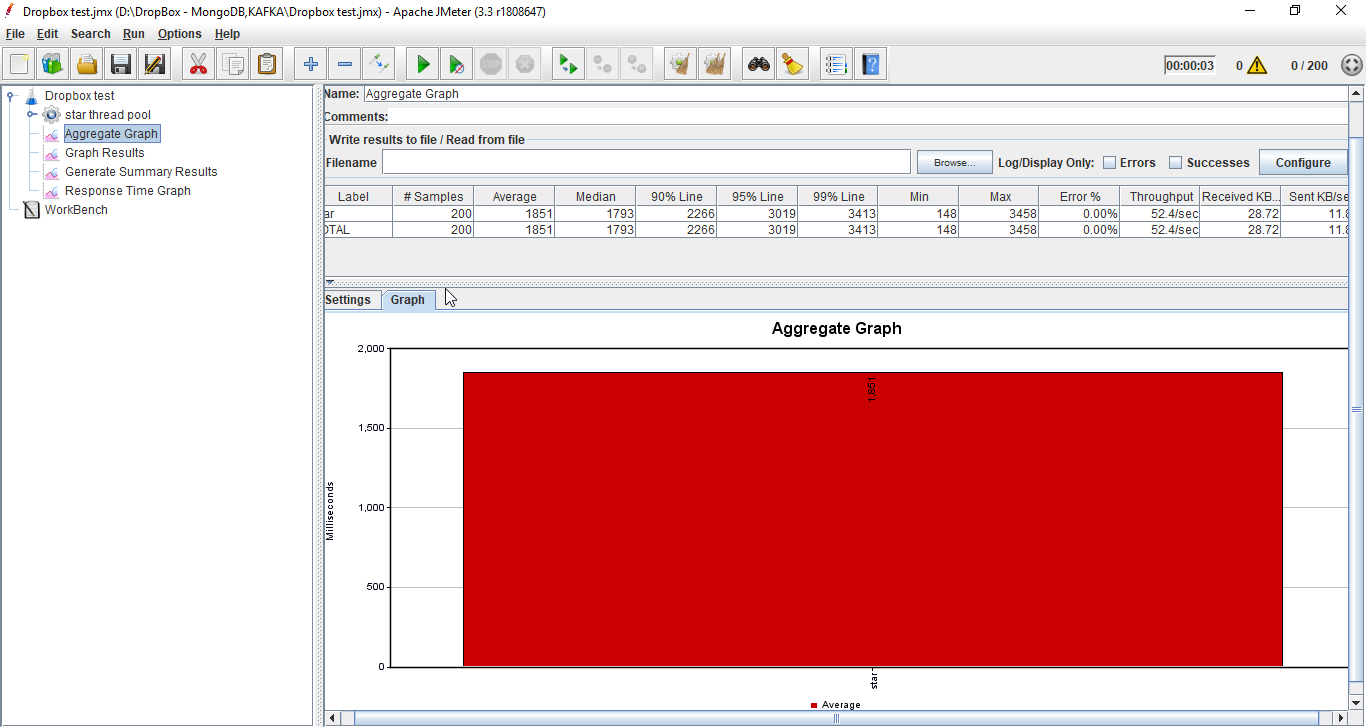


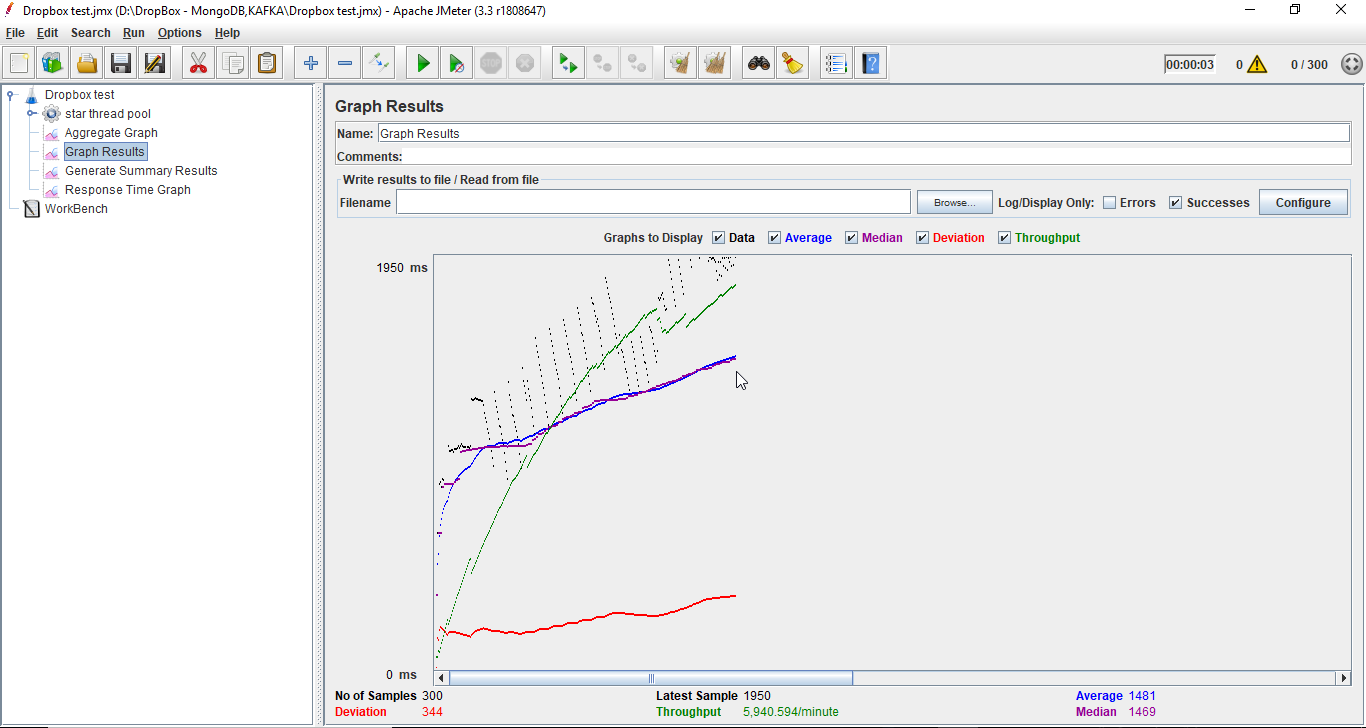
**Without Connection Pooling**:



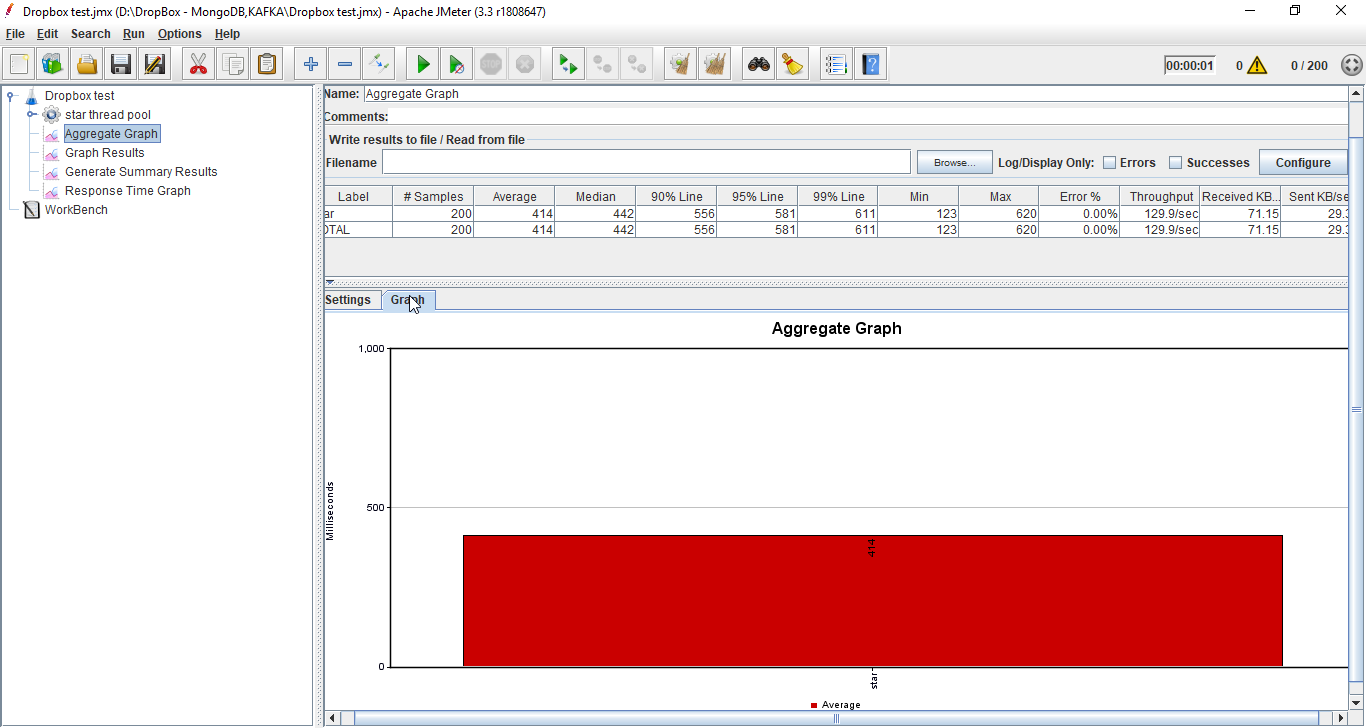


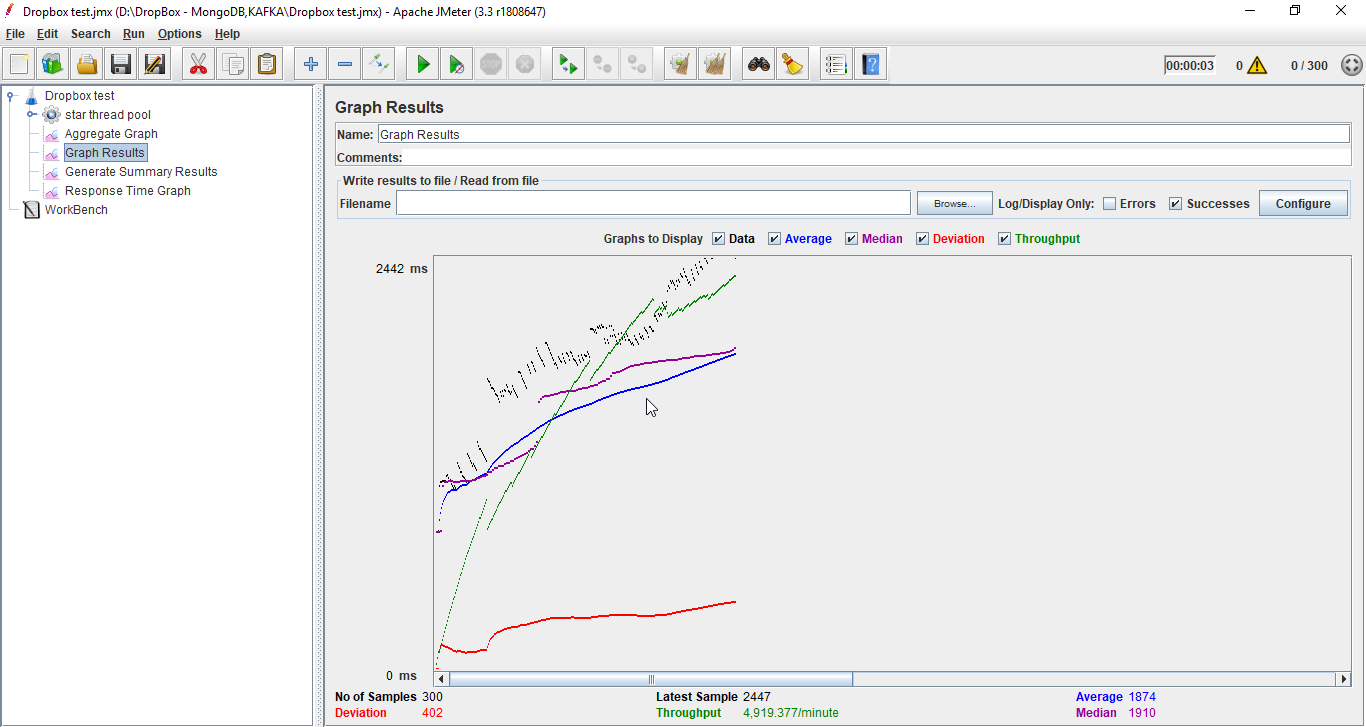
**With Connection Pooling**: DB provided



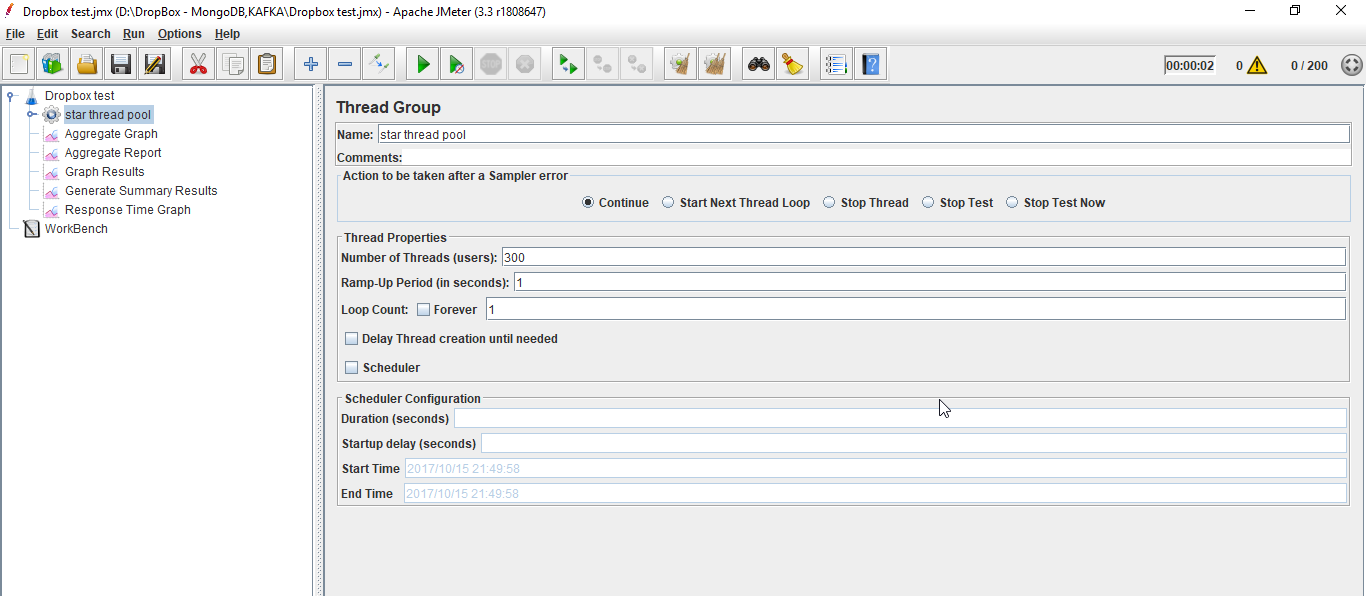
****

**With Connection Pooling**: own connection pooling implementation

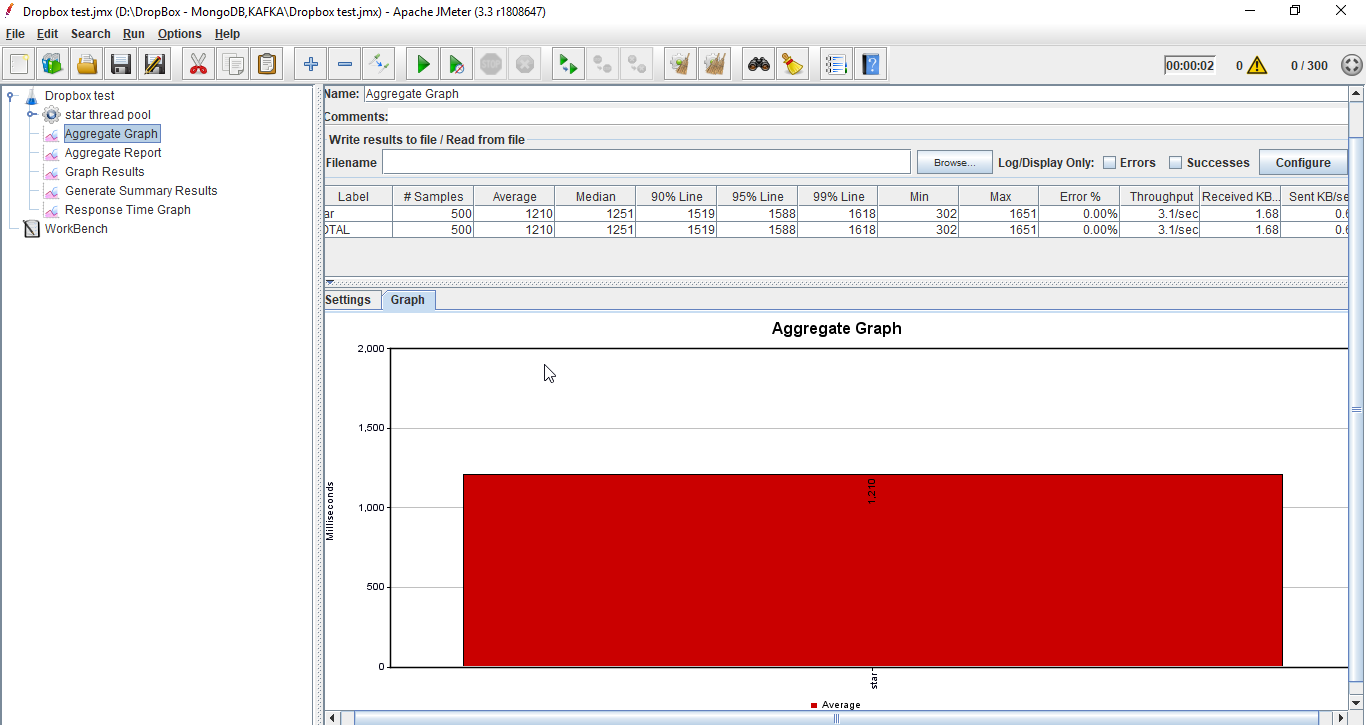


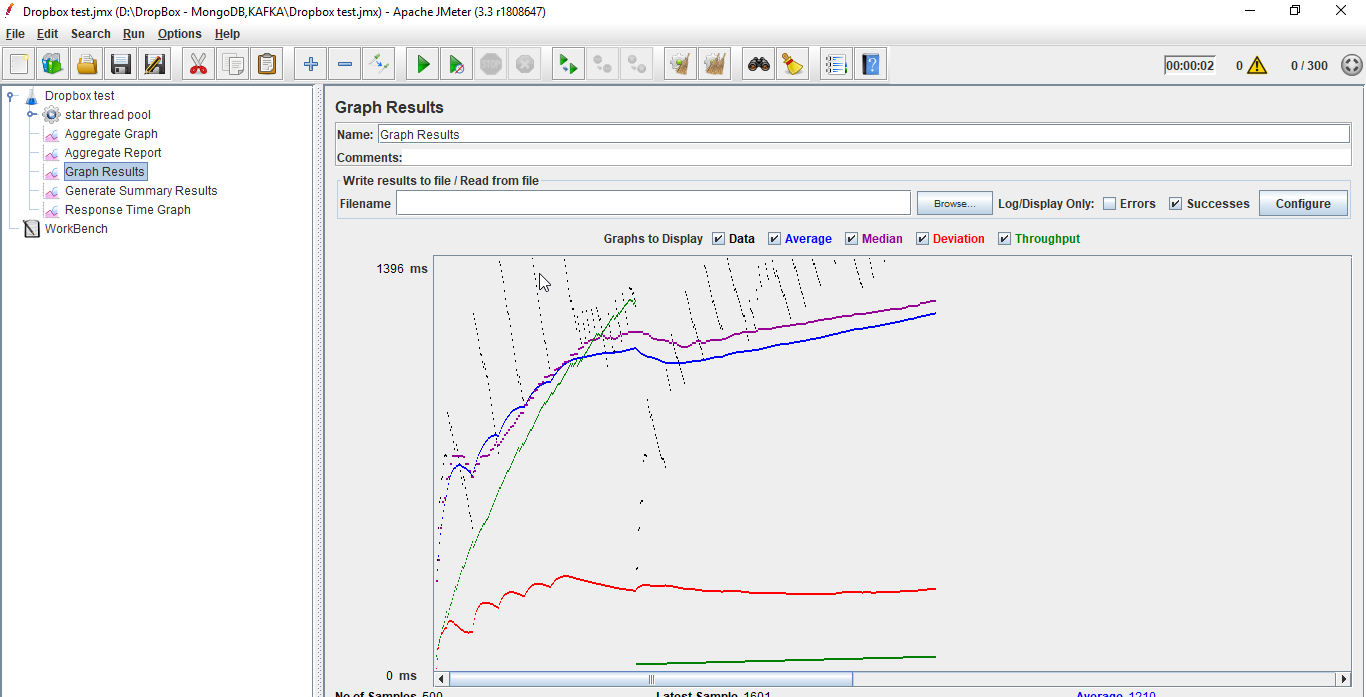


1. **For 300 concurrent users**:  
   **Setup**-

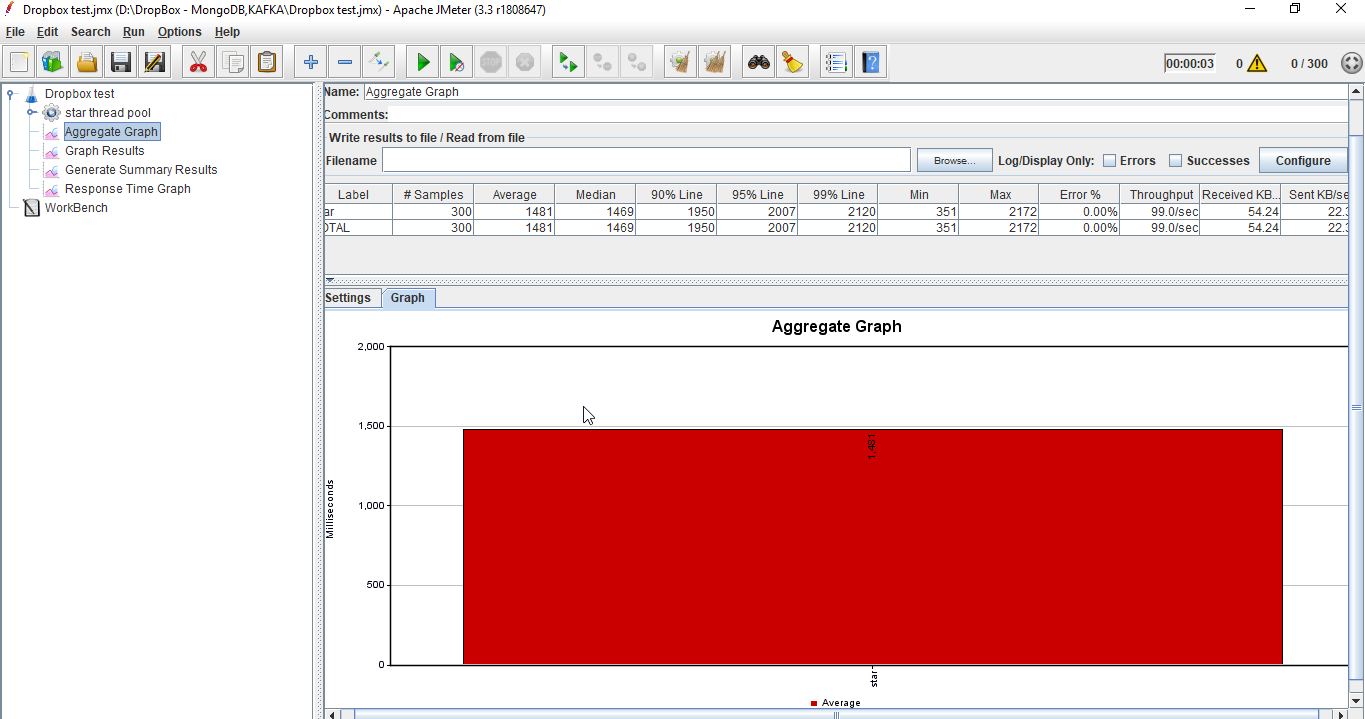


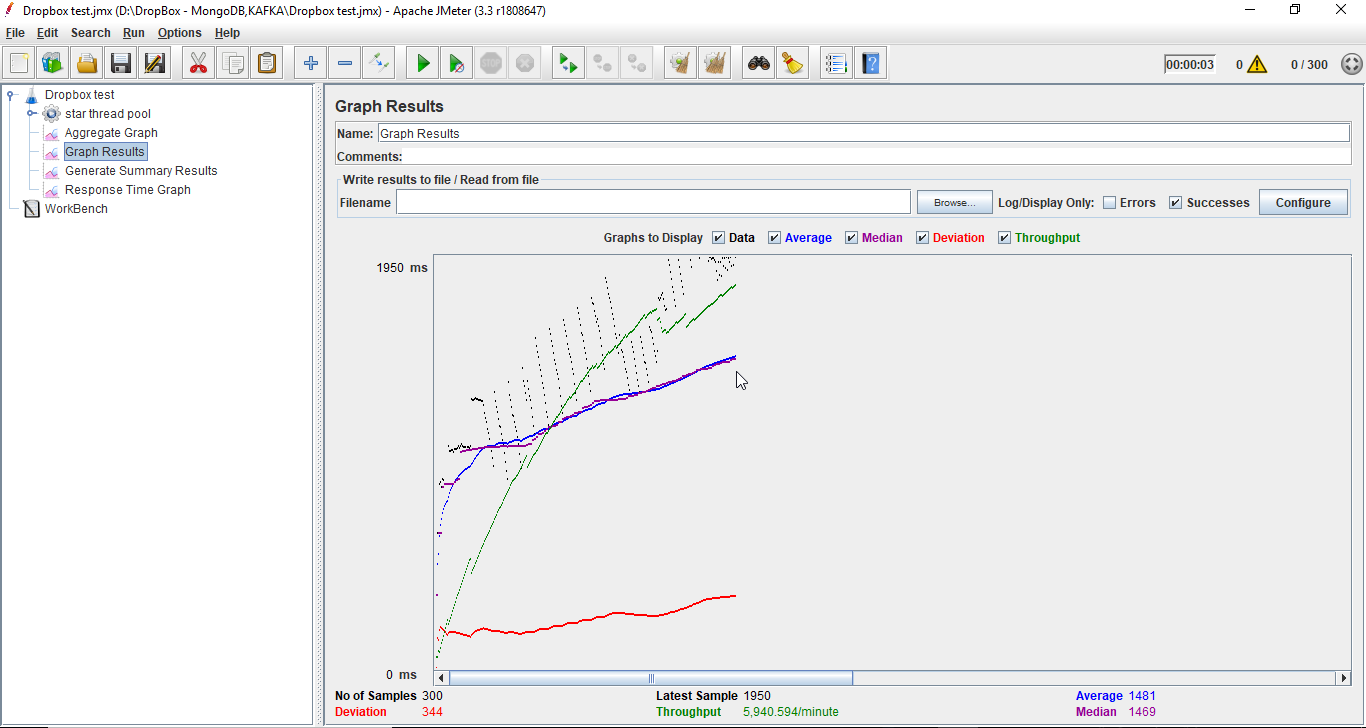
**Without Connection Pooling**:



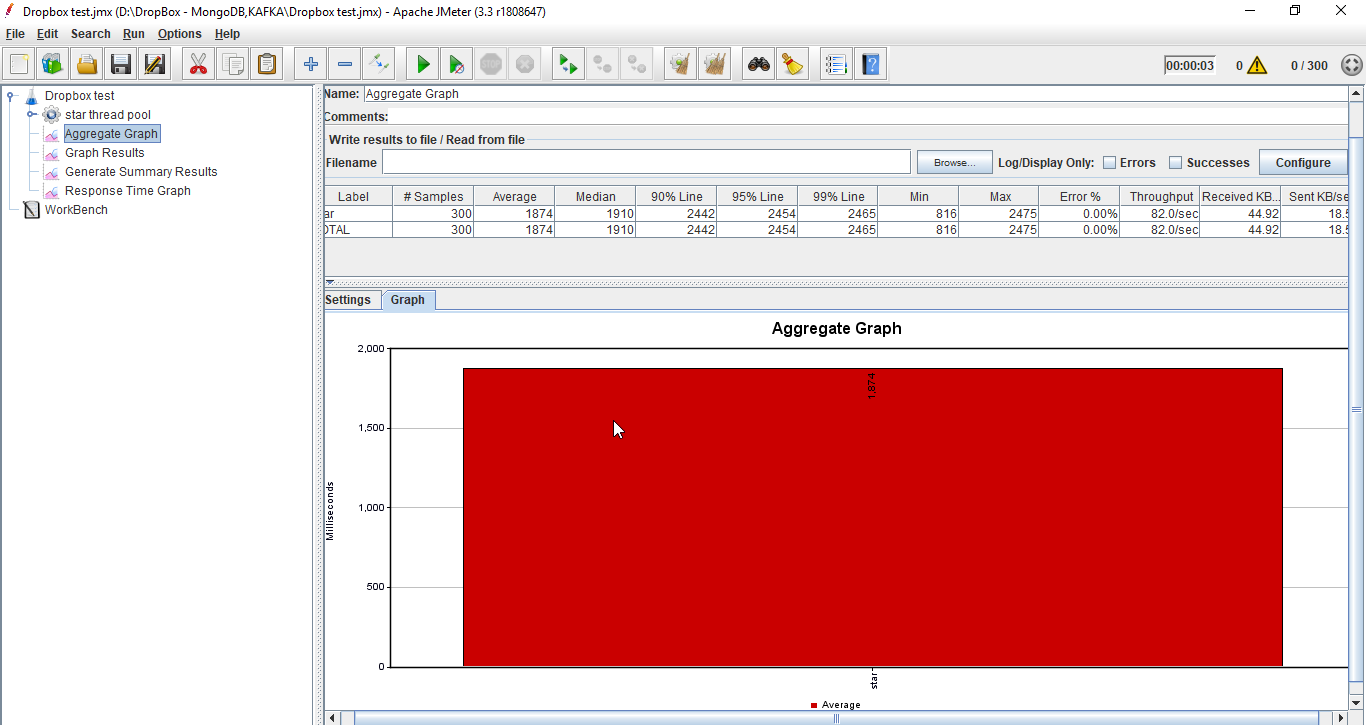


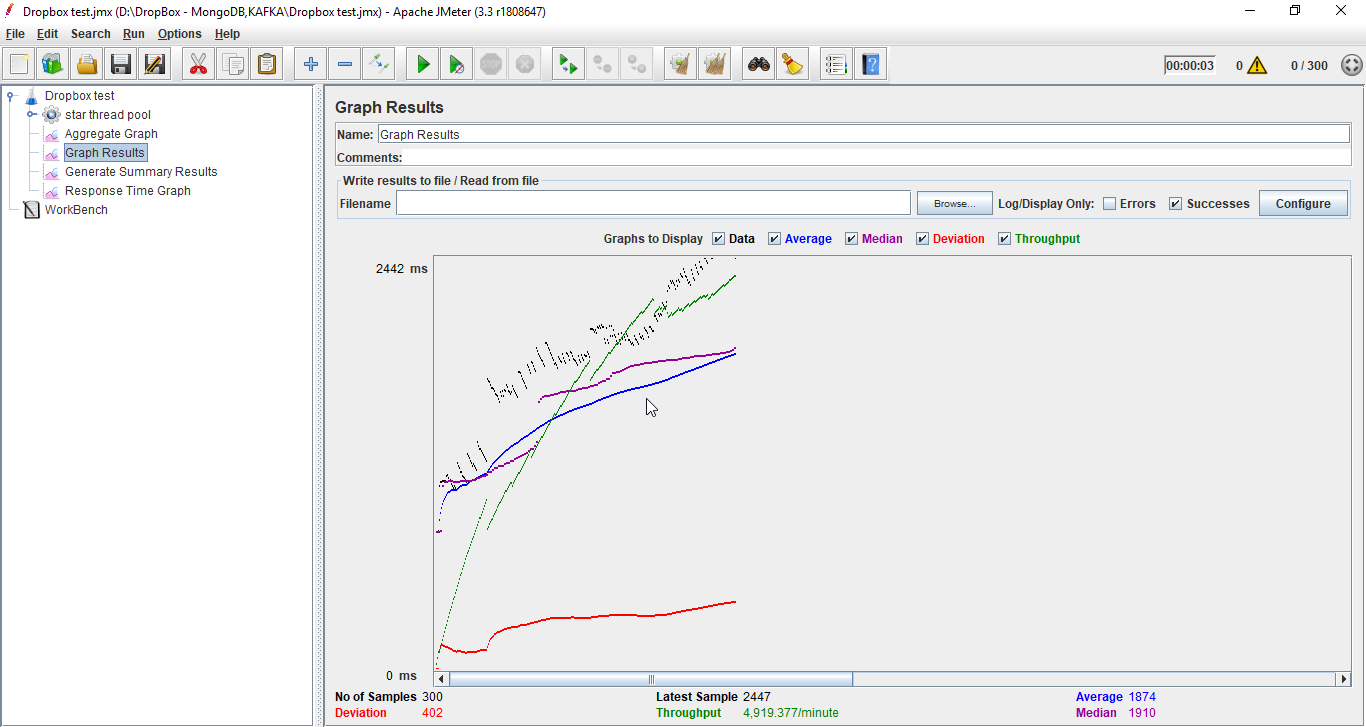
**With Connection Pooling**: DB provided



****

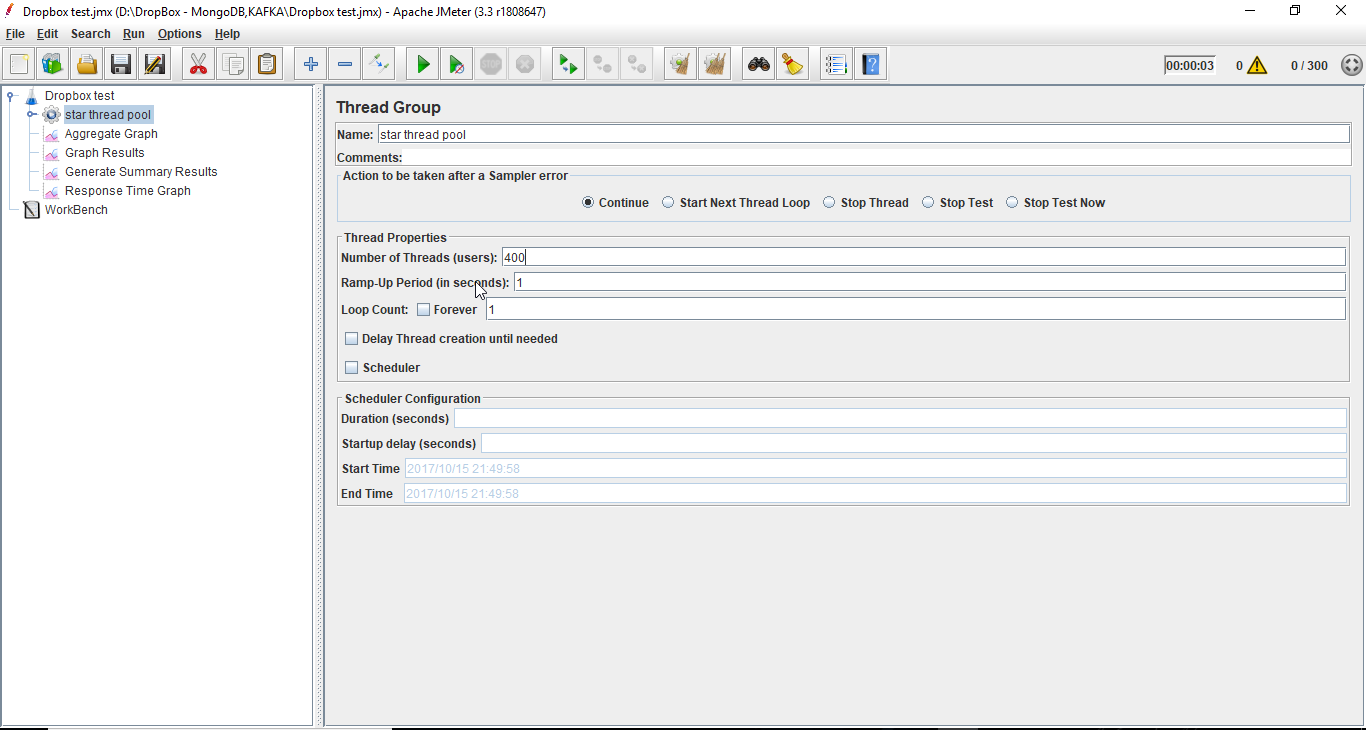
**With Connection Pooling**: own connection pooling implementation



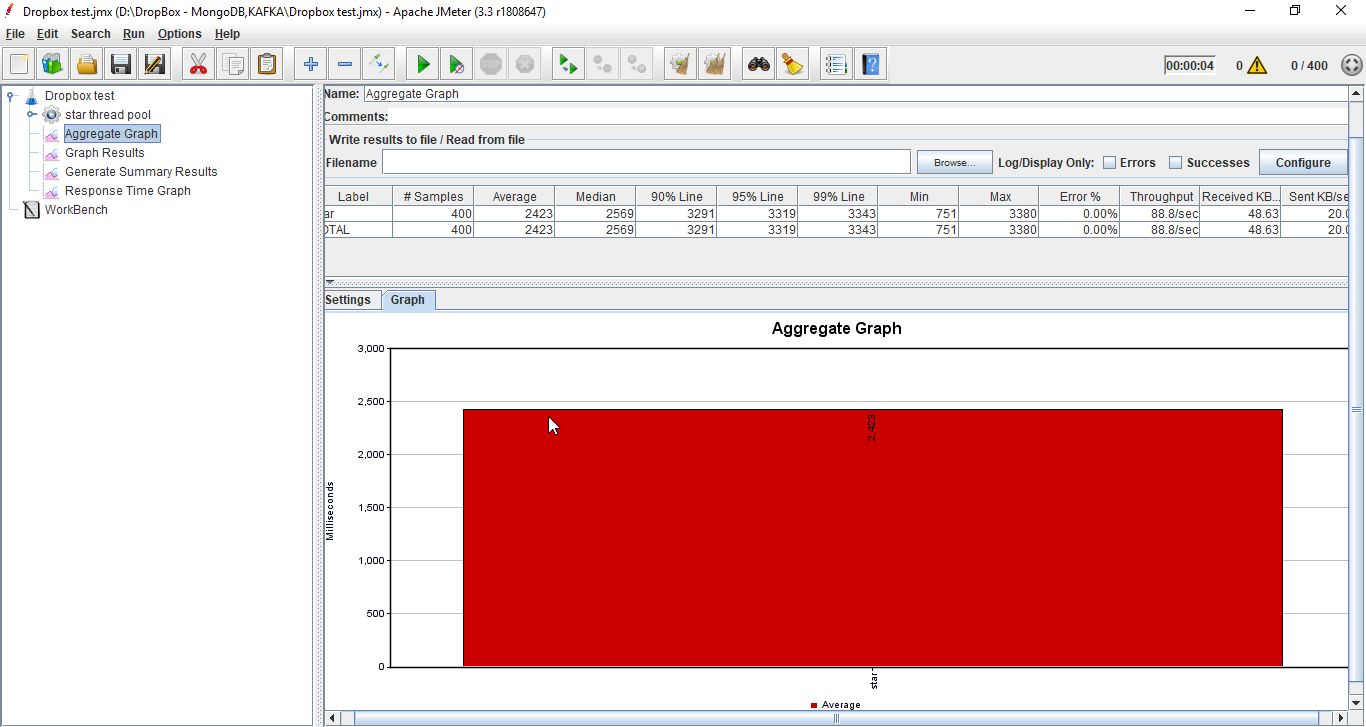


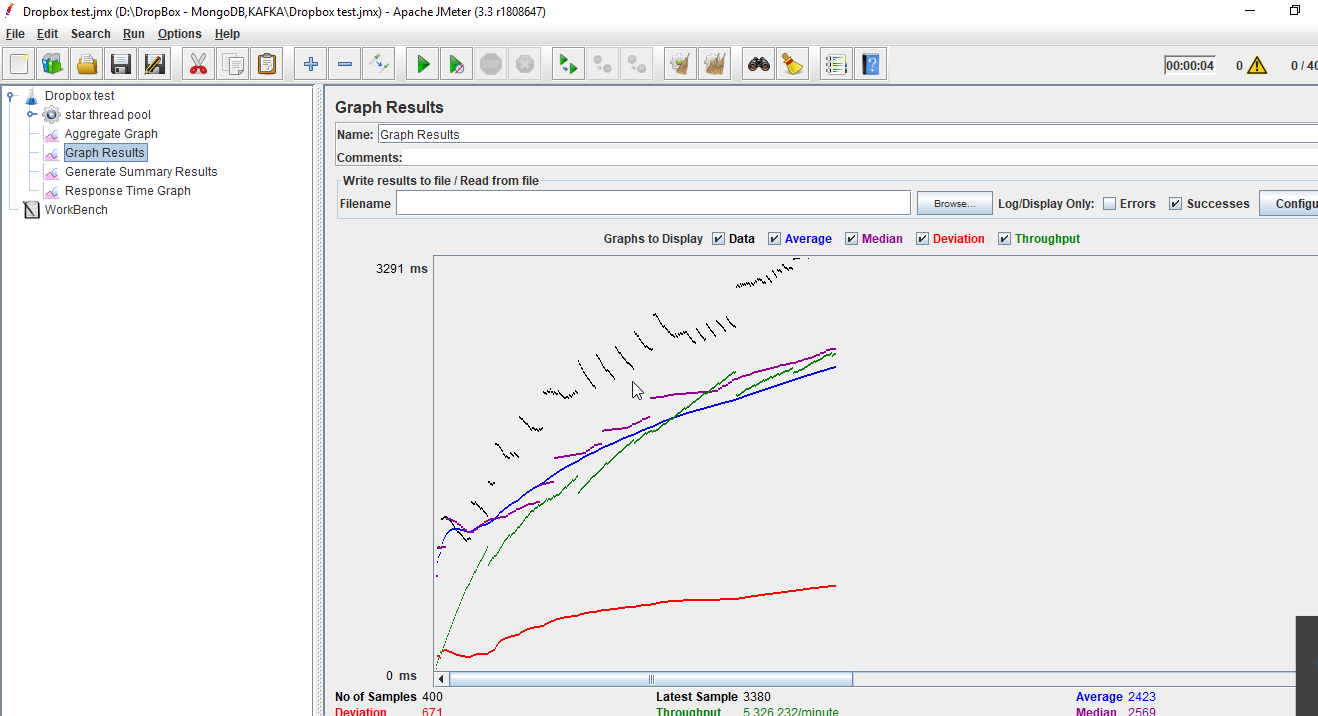
1. **For 400 concurrent users**:

**Setup** –

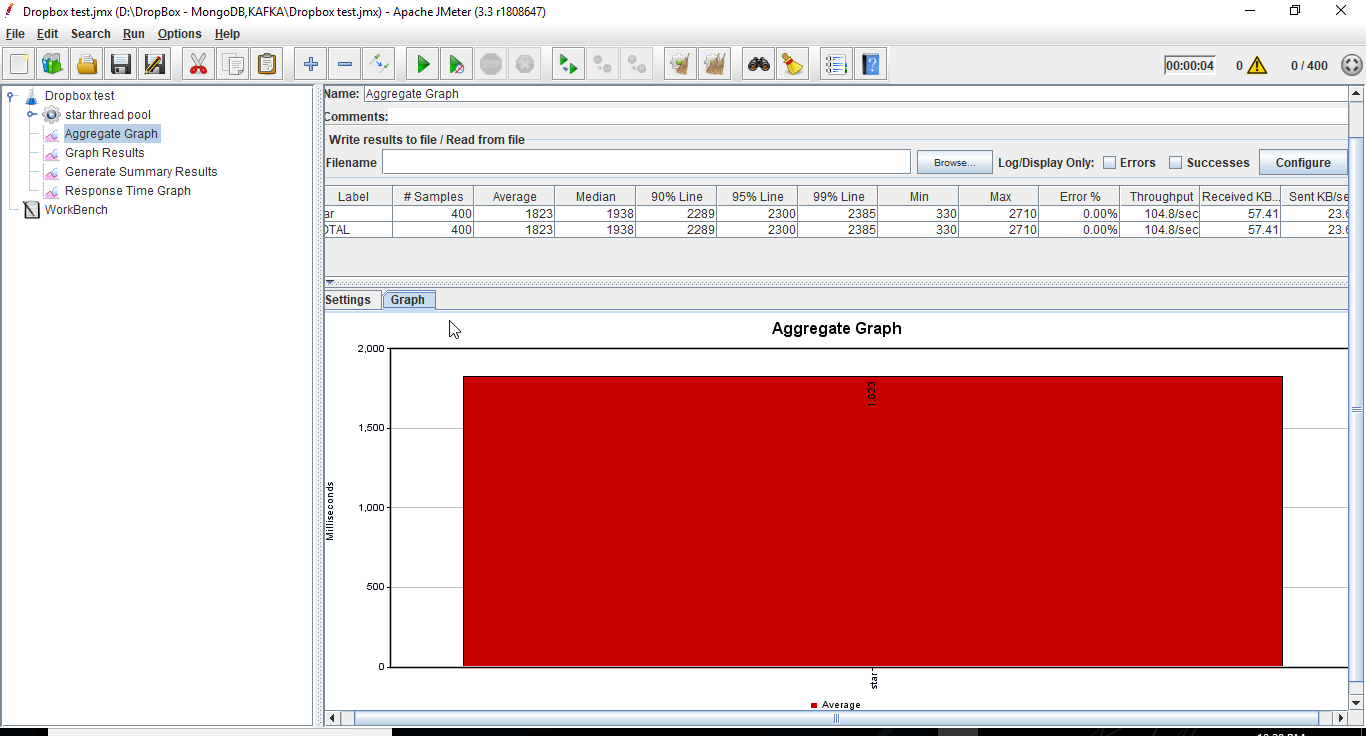


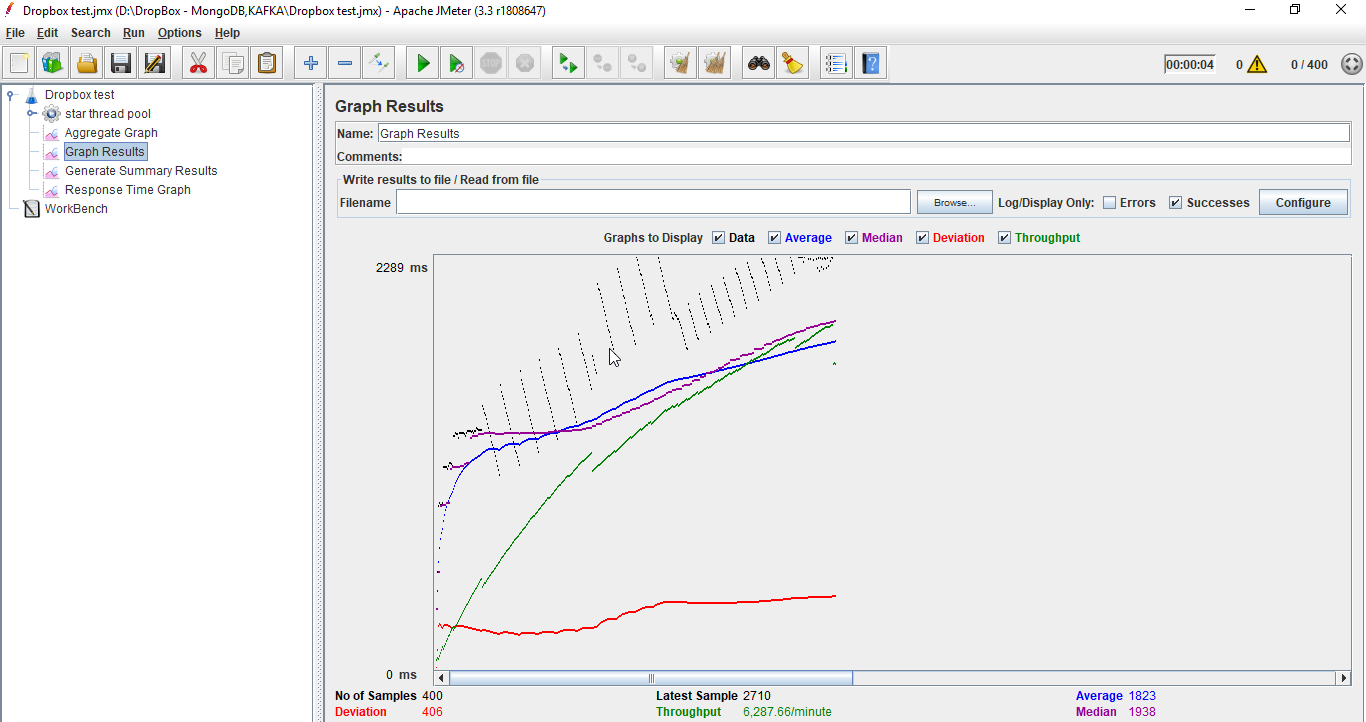
**Without Connection Pooling**:



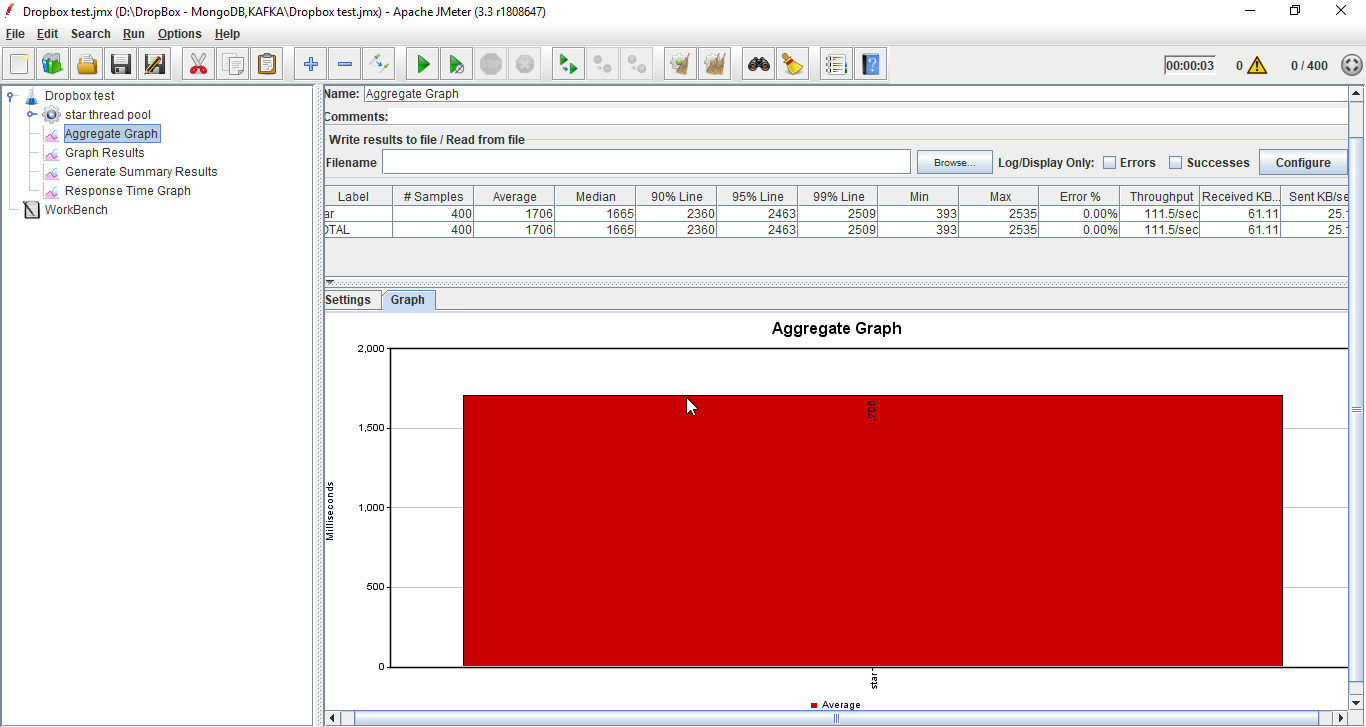


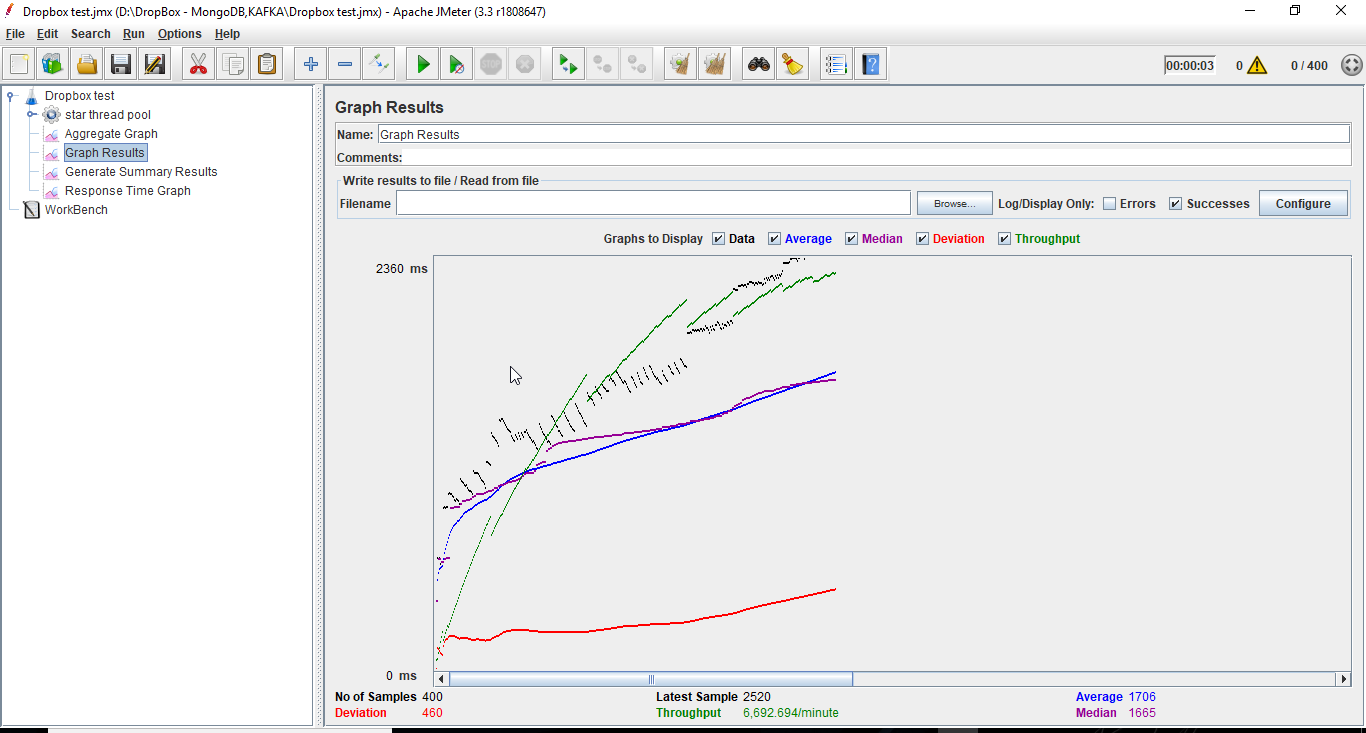
**With Connection Pooling**: DB provided



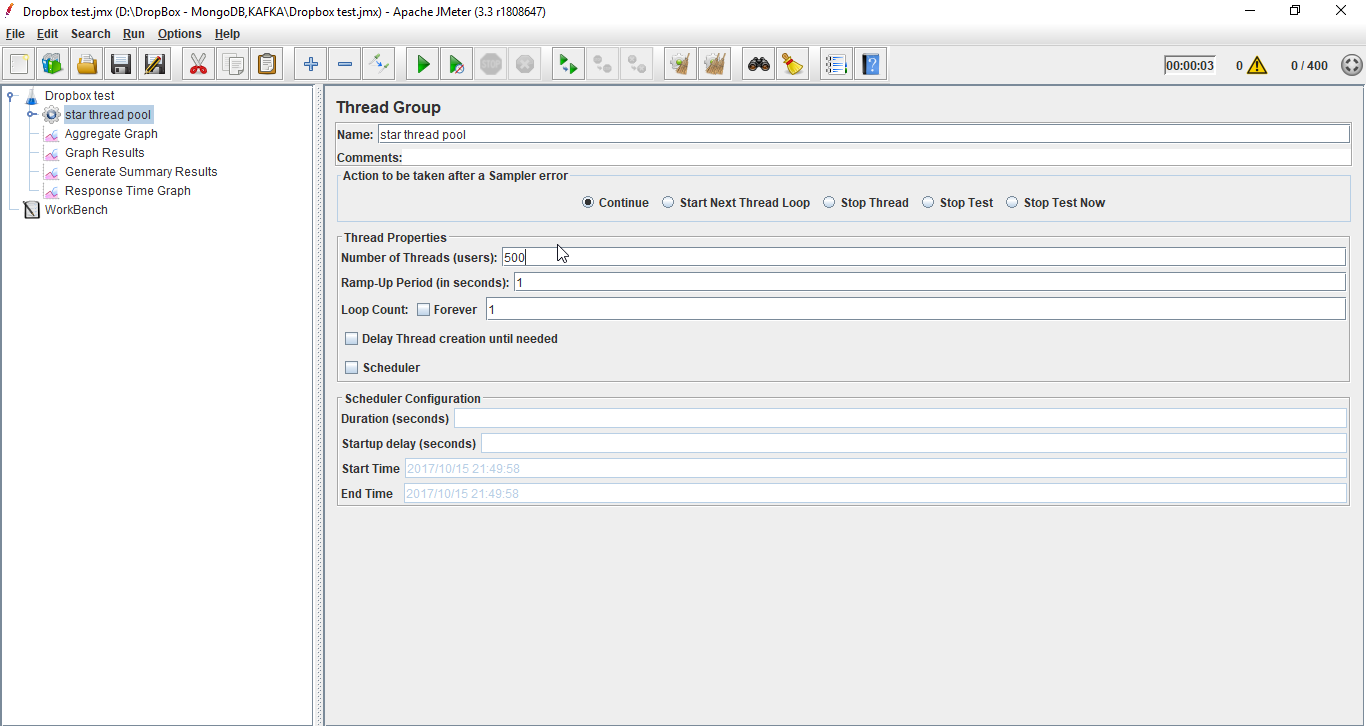
****

**With Connection Pooling**: own connection pooling implementation

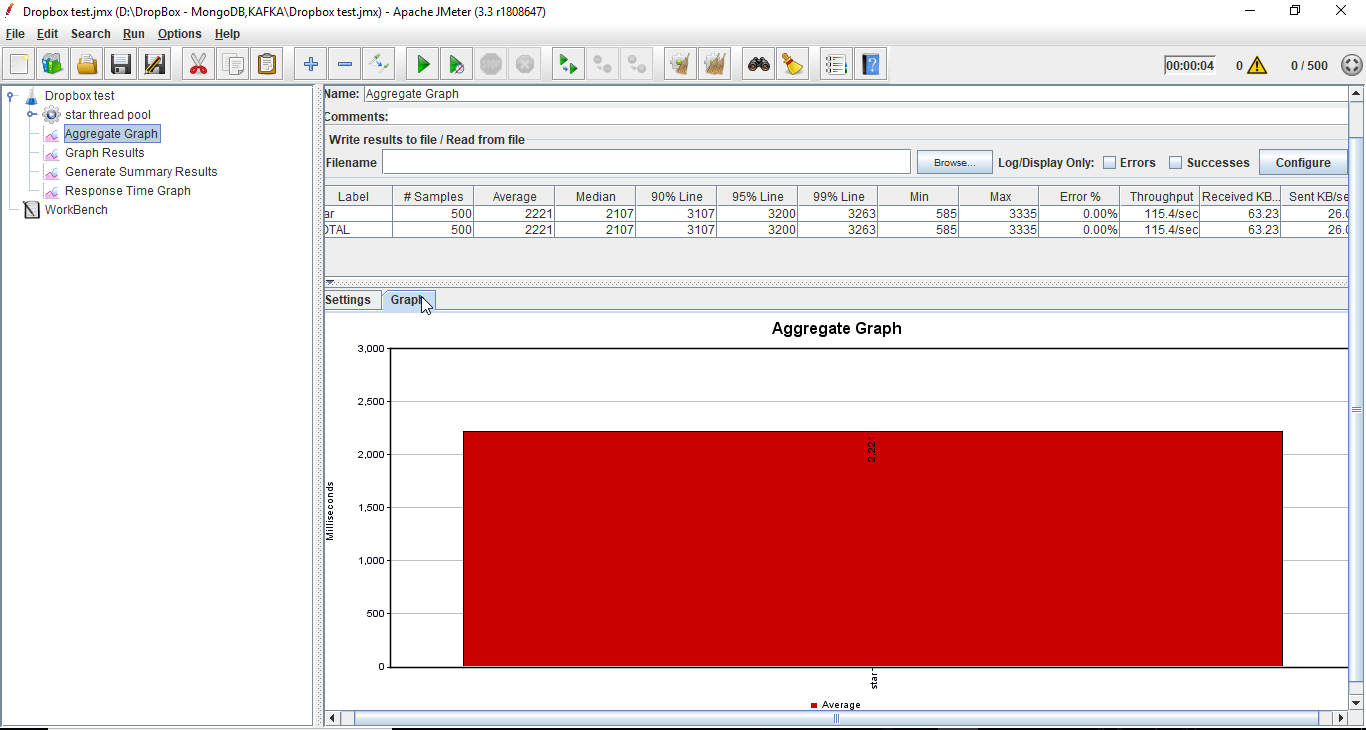


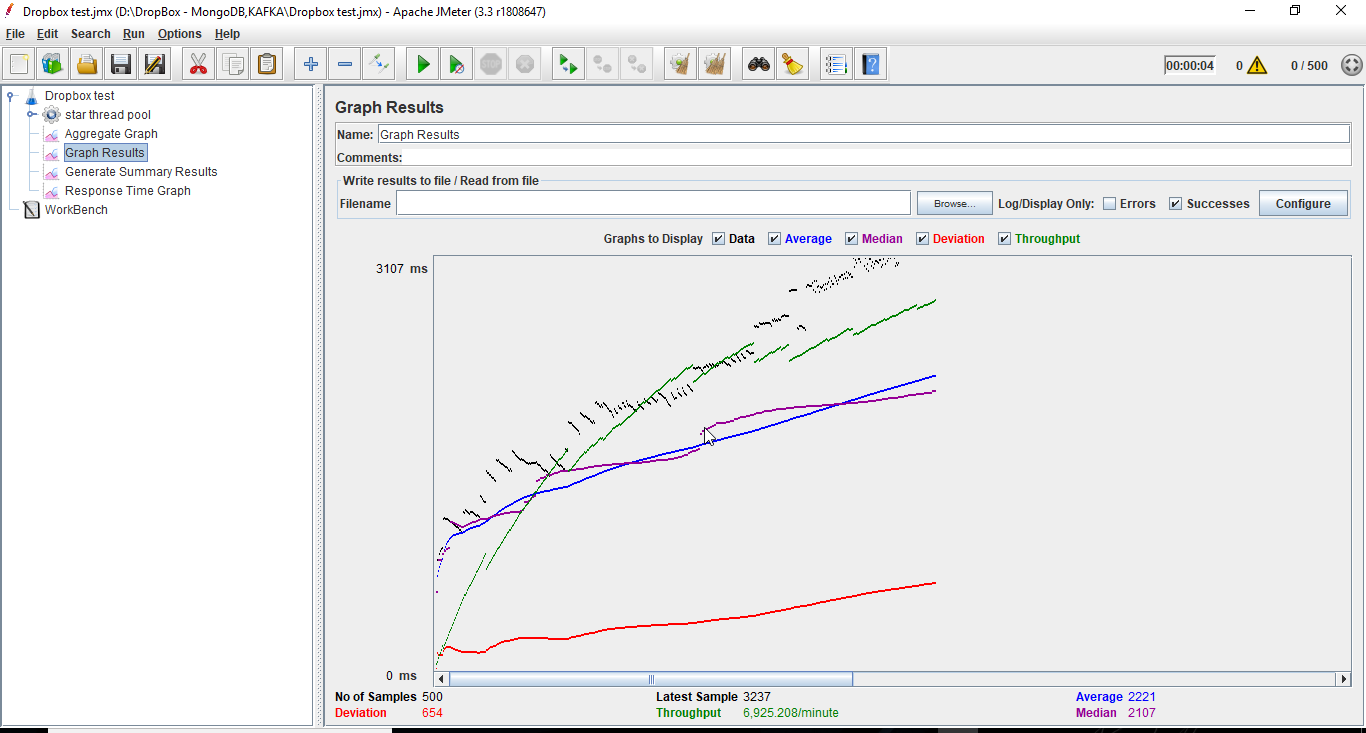


1. For 500 concurrent users:  
   **Setup** –

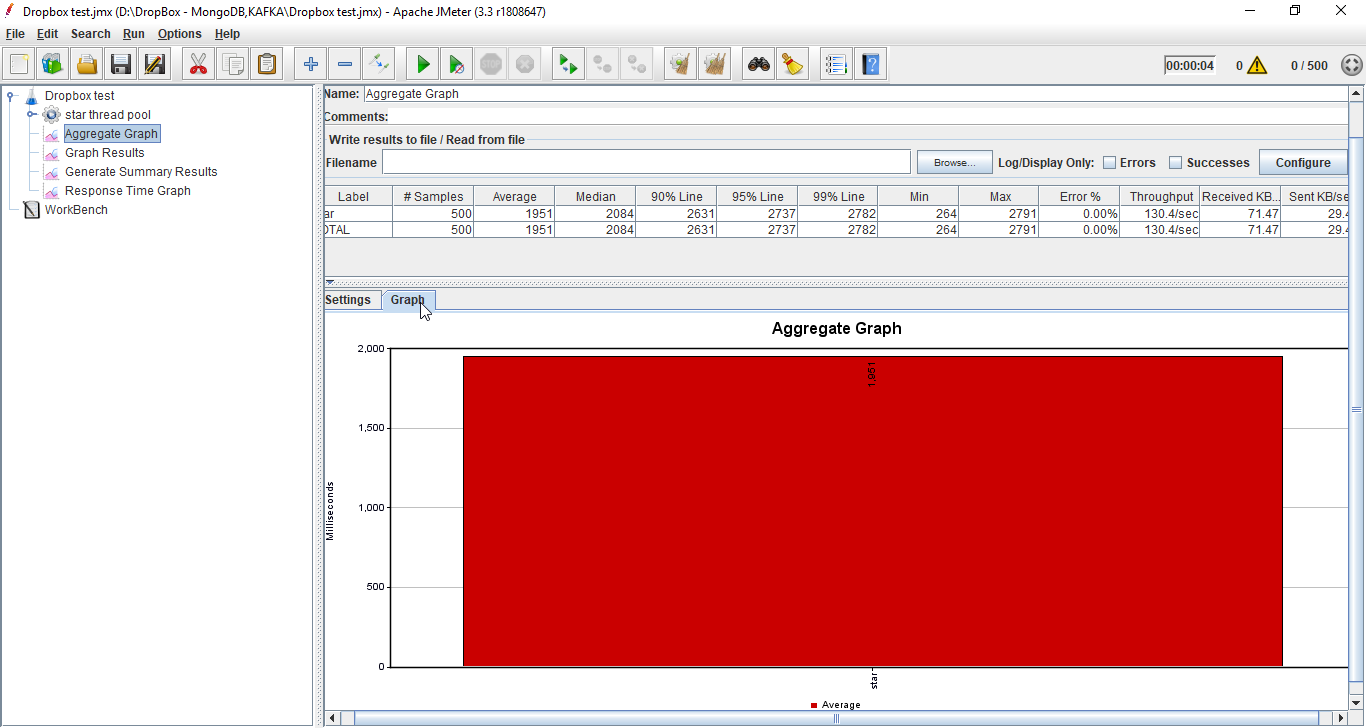


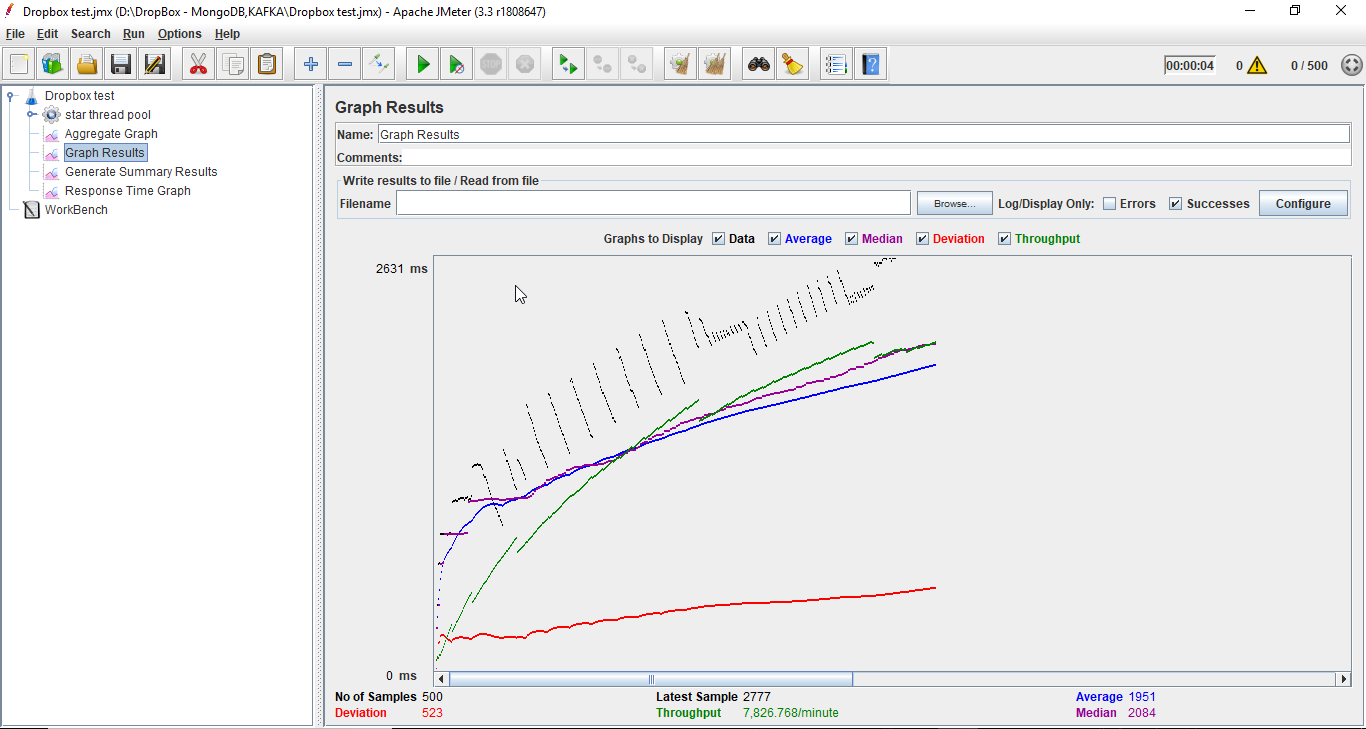
**Without Connection Pooling**:



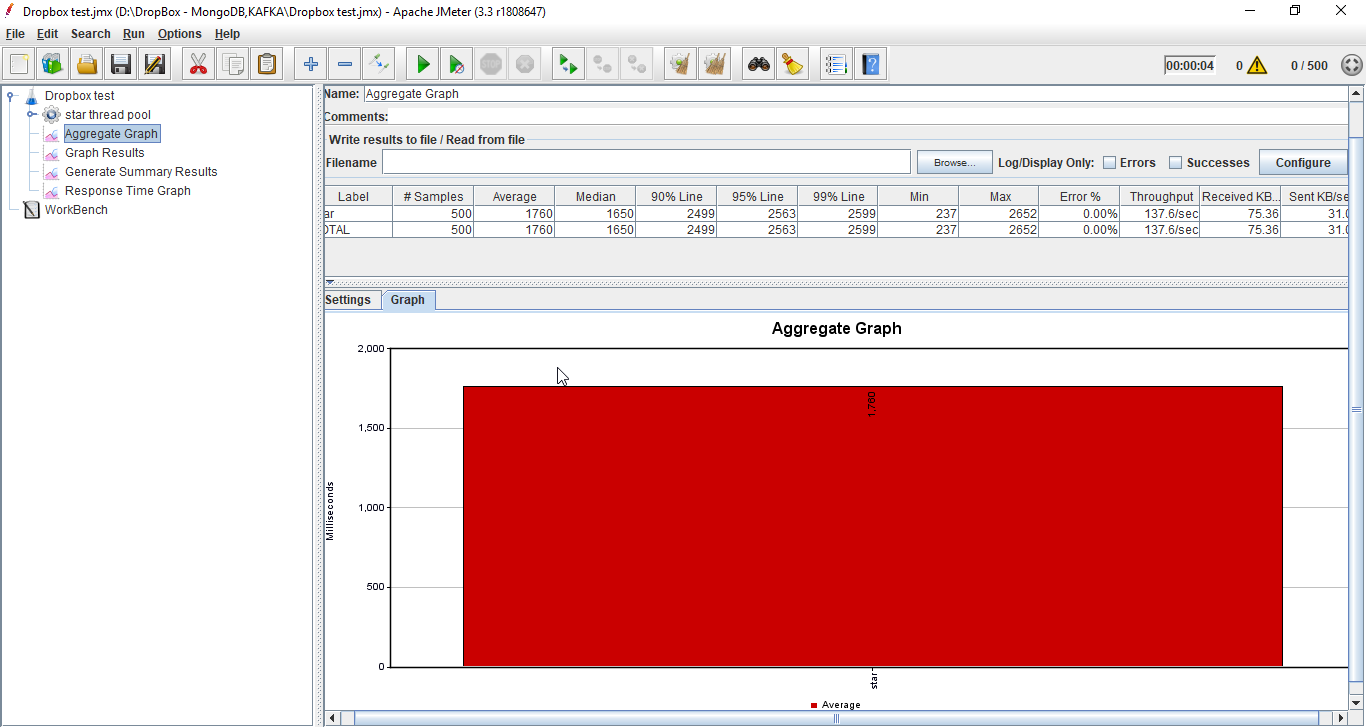


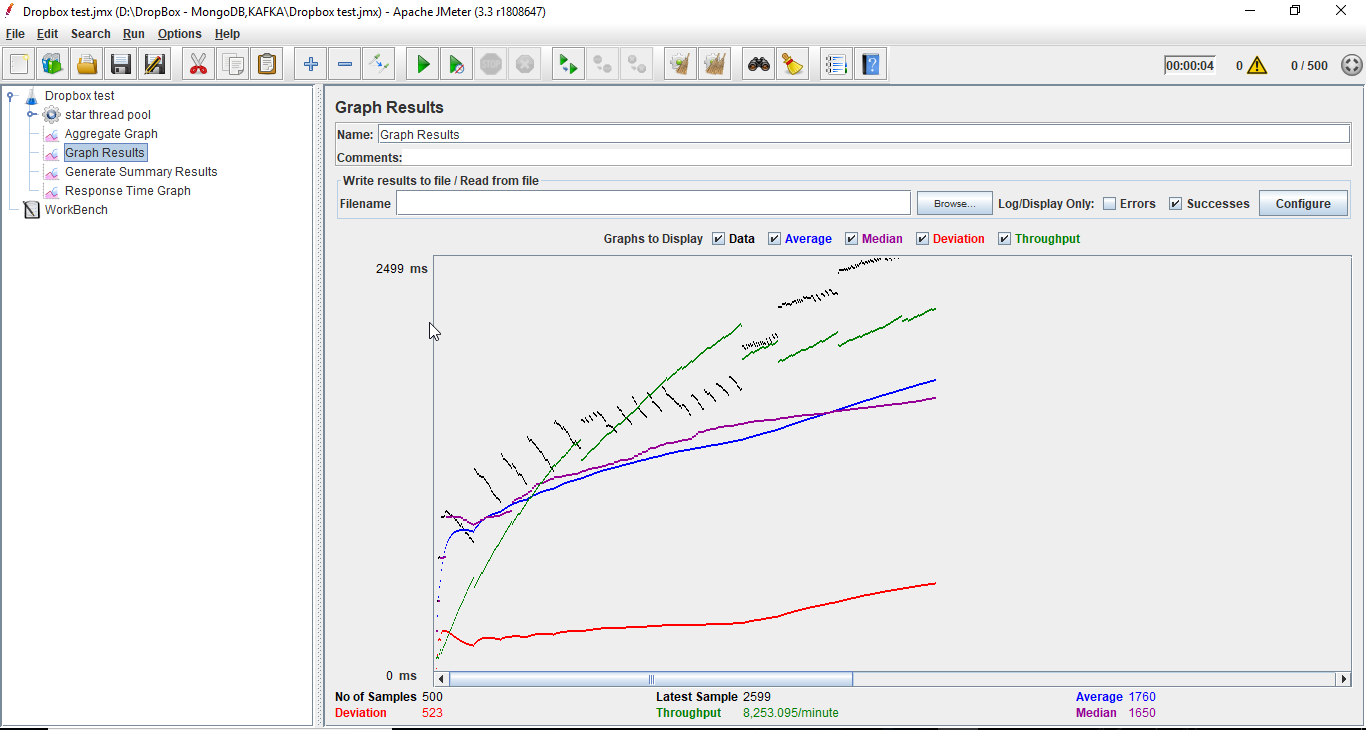
**With Connection Pooling**: DB provided



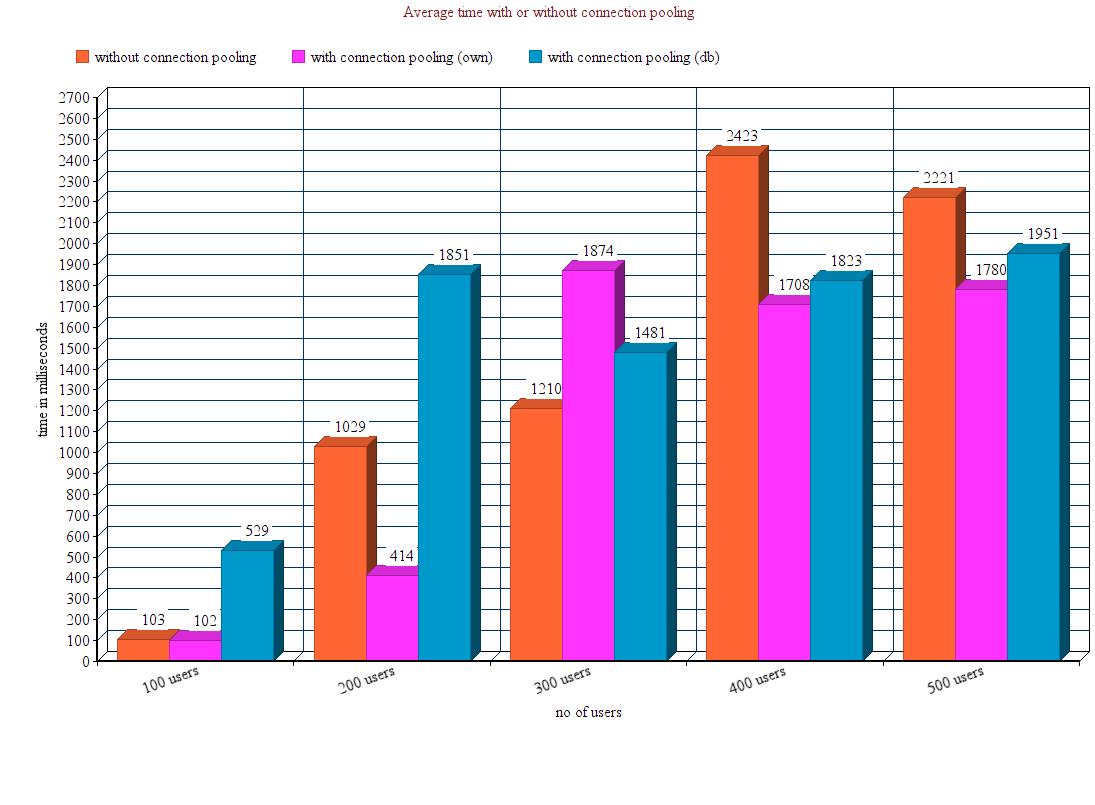
****

**With Connection Pooling**: own connection pooling implementation





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



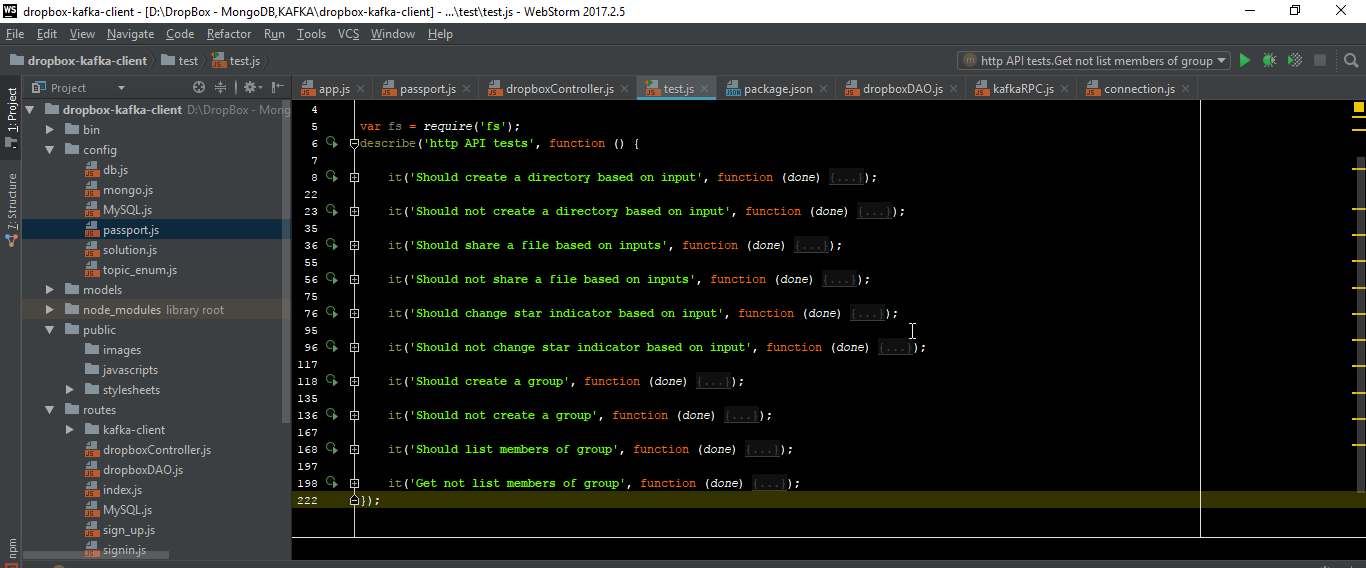
### Analysis of Performance testing

It can be observed from the graphs shown above that if the connection pooling is implemented in the application then the average time for request and response decreases to a considerable amount. When multiple concurrent users are requesting for resource; then connection pooling will be very useful because it reduces the number of times new connections are created. Therefore, the process of getting a connection also becomes faster.

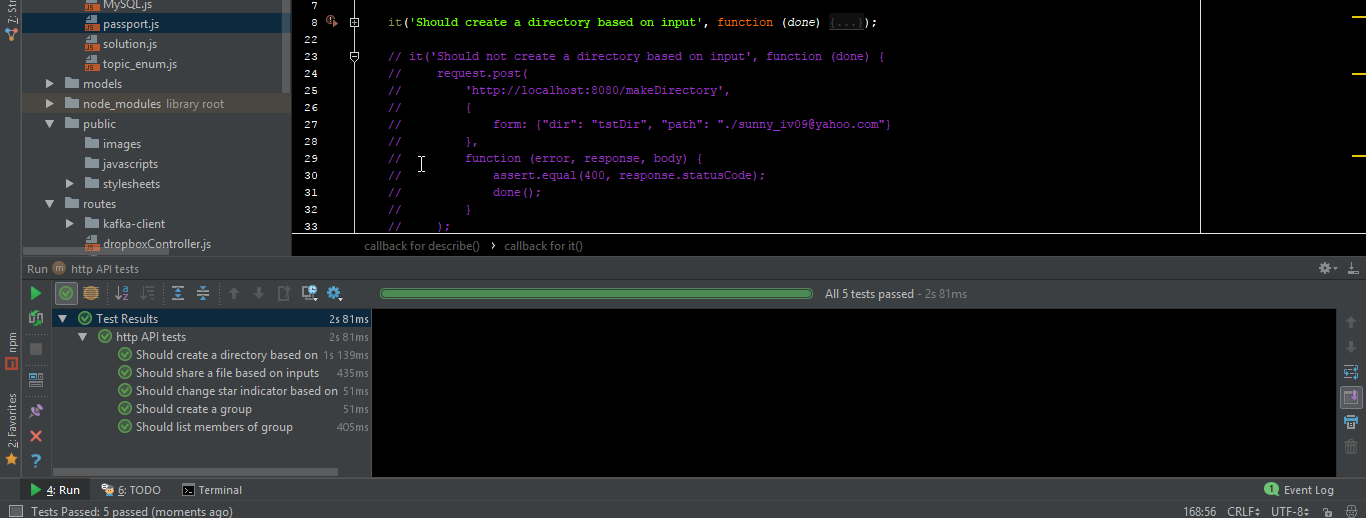
In my application I have made a pool which allocates connections to database. At present, I have kept the pool size 10. When a new connection is needed it will simply borrow the connection from the pool and release as soon as the work gets completed

## MOCHA TESTING:

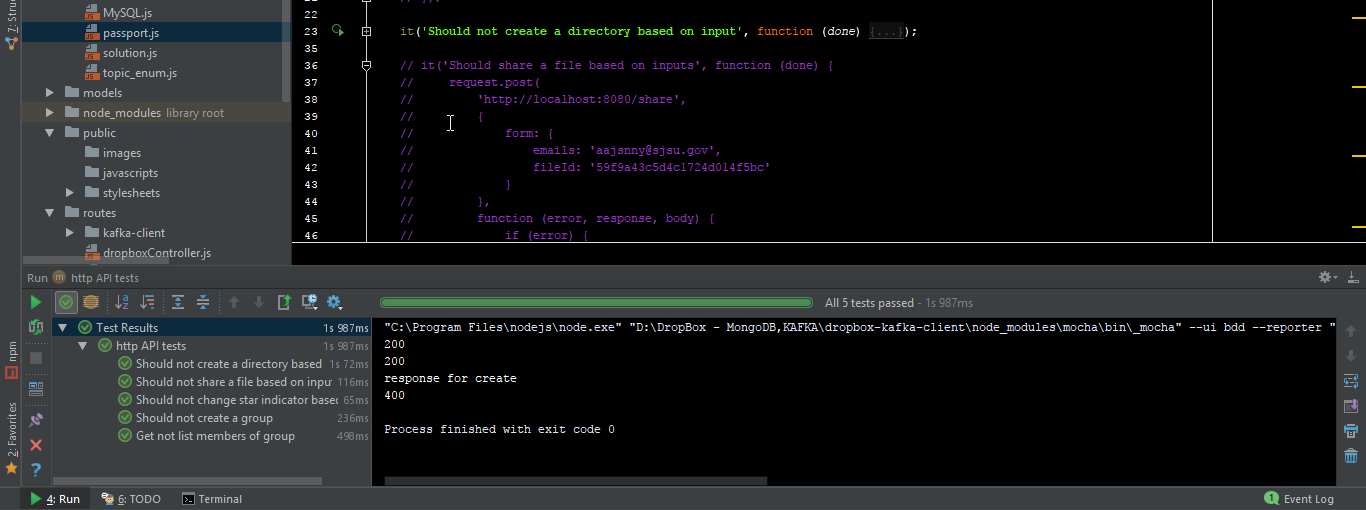
I have tested 10 API calls using MOCHA.

****

**Positive Test cases:**

****

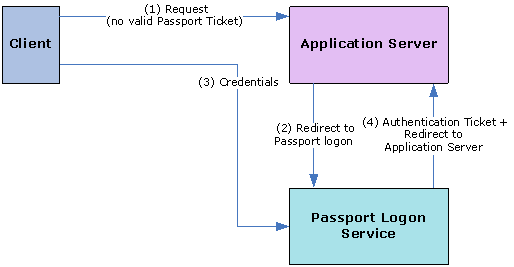
**Negative Test cases:**

****

# **Lab Questions**

1. *Compare passport authentication process with the authentication process used in Lab1.*

Passport authentication is a centralized authentication service/middleware for Node used solely to authenticate requests. Since each application has its own authentication requirements, traditional authenticate services fails to recognize that. However, in Passport, there are strategies that can be employed without creating unnecessary dependencies.



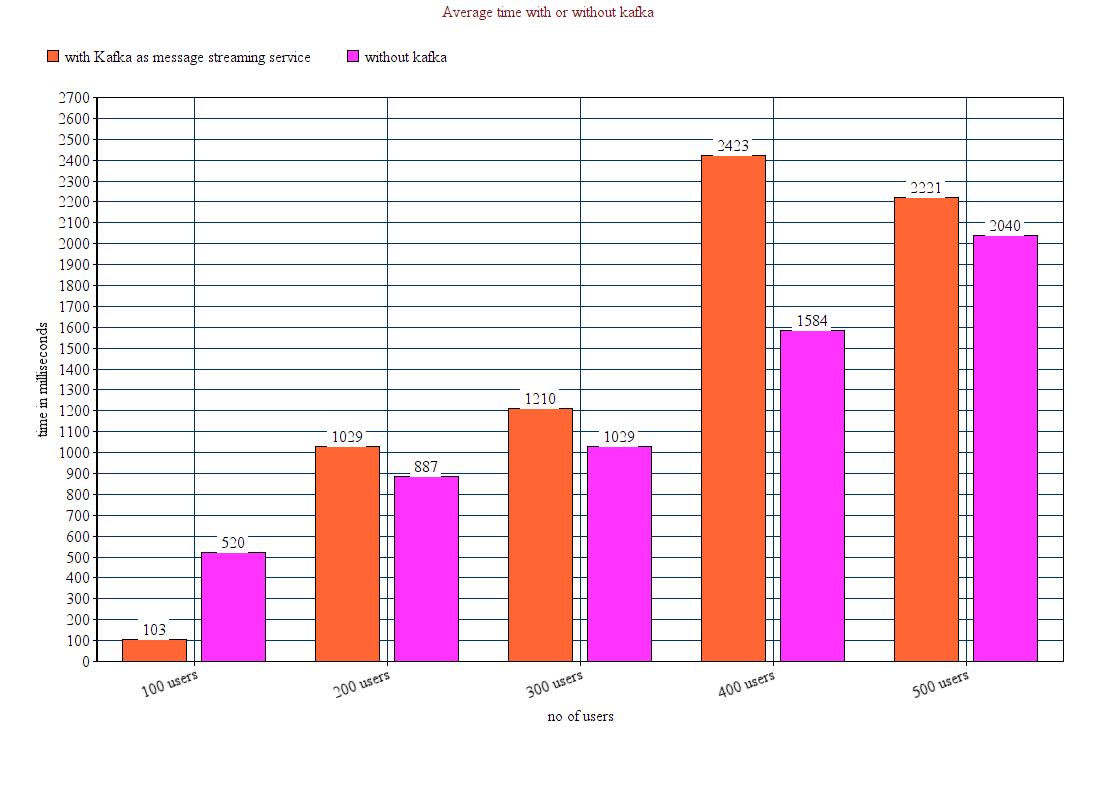
As compared to established authentication processes, coding with Passport does not seem to be complicated. The authenticate request is as simple as calling “passport.authenticate()”. Unlike other approaches, it is convenient to use authenticate()’s function signature as route middleware in Express applications.

exports.validateLogin = function (*req*, *res*) {  
  
 ***console***.log(**"passport call"**);  
 passport.authenticate(**'login'**, function (*err*, *results*) {  
 if (*err*) {  
 ***console***.log(*err*);  
 *res*.status(500).send();  
 }

After successful authentication, Passport will establish a persistent login session. However, other traditional approaches failed to do that. Moreover, if the built-in options are not sufficient for handling an authentication request, Passport manages a custom callback, which allow the application to handle success or failure.

In our lab, we authenticated requests on our own by hashing passwords, which is yet another characteristic of passport. The login form is submitted to the server via the POST method. Using authenticate() with the “local” strategy will handle the login request, allowing the separation of extremely critical module of authenticating a user and maintain code easily.

1. *Compare performance with and without Kafka. Explain in detail the reason for difference in performance.*



The graph above shows the comparison between the applications’ average response time when it is using Apache Kafka as message streaming service and when it doesn’t.

While applying mentioned scenario in both with kafka and without kafka, it seems that average time while using kafka is higher than without kafka. Because, in this development scenario there are multiple topics, but with only one partition. When kafka is deployed on different servers with multiple partitions, it is most effective as it is highly efficient to send and receive messages to respective receivers.

Also as a large number of messages are added to the queue very quickly, kafka consumers still process single message at a time from the queue and ensure proper handling of all the requests.

We can say a trade-off between reliability and performance happens when adopting any distributed messaging service like Apache Kafka when running application on a single server.

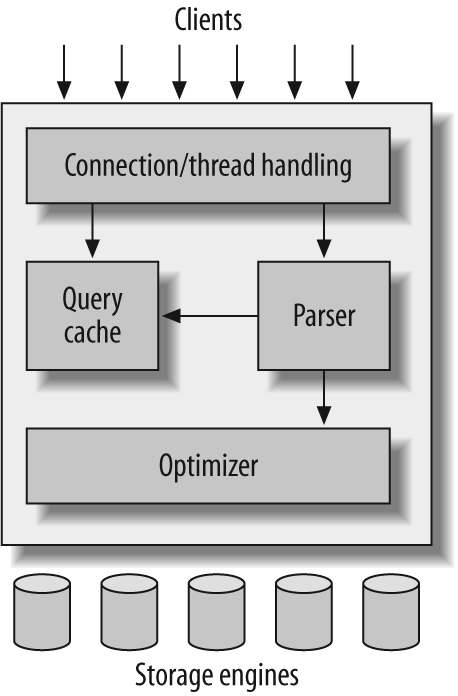
In a scenario where the application is highly distributed among multiple nodes and partitions, kafka will definitely provide more throughput and better reliability.

1. *If given an option to implement MySQL and MongoDB both in your application, specify which data of the applications will you store in MongoDB and MySQL respectively.*

SQL is well-recognized for its high performance, flexibility, reliable data protection, high availability, and management ease. Proper data indexing can solve the issue with performance, facilitate interaction and ensure robustness. However, if your data is unstructured and complex, or if you cannot pre-define your schema, you would better opt for MongoDB.

Both databases have their own pros and cons, for my application I would save the most critical data like user details, user activity logs, group details,etc in MySQL as it is a general solution providing transaction control, administration facilities whereas MongoDB is reliable for storing large quantities of data and when structure of data is complex. So I would use MongoDB for dumping user files, videos, music, etc. which provides inbuilt support for performing Map-Reduce and horizontal scalability by sharding.

MySQL architecture



MongoDB

