**DROPBOX – WITH KAFKA AND MONGODB**

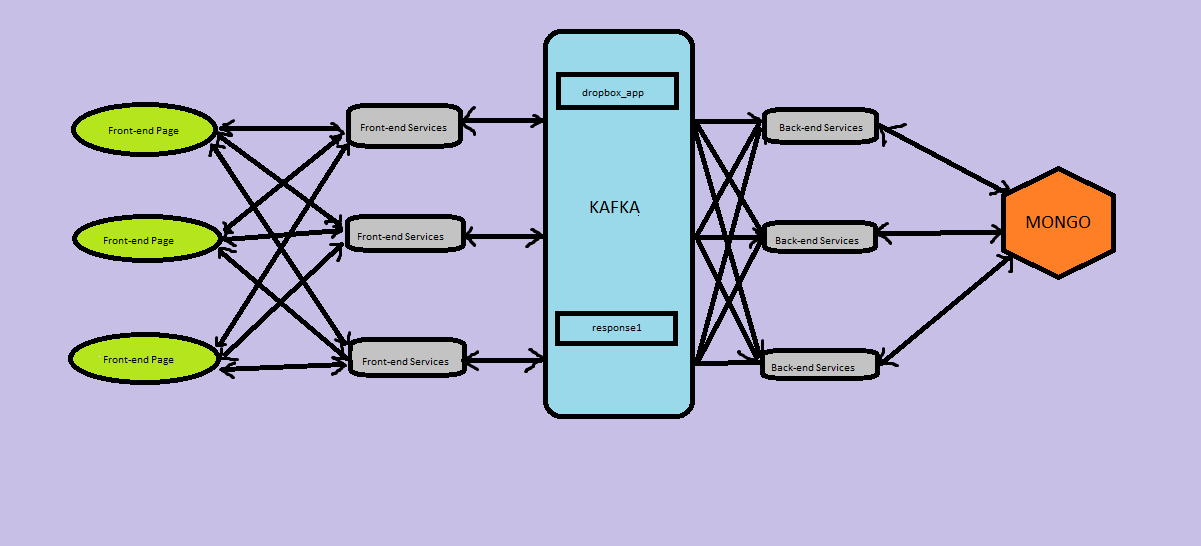
*Introduction*

The application is a simulation of a Dropbox application which will be used to store files and folders on the server for a user for which he/she can share with groups and individual and use it for a later time.

The purpose of the system is to securely store the user files on the server which they can use in future, maintain their files and use it online wherever they require.

The application is built with 3 servers running in parallel, namely front-end, node server and Kafka node server.

*Architecture and System Design*



The React Front-end interact with the Node Server, the Node server checks for Authenticity and then allows the data to flow forward.

The Node server calls the Producer to send data to kafka for specific topic and store in it. The Node Back-end server continuously listen on the specific topic and then it calls corresponding function for the operations to perform and return back data.

The Kafka back-end server send the response data to kafka and the node server continuously listen to another topic and retrieve the data.

The data is then given back to redux which is then shown/download on the front-end.

The Front-end is designed with React-Redux, HTML5 and BootStrap 3.7, while backend features Node JS as the Backend Server, Express as Scripting Language in JS and MongoDB Database implemented with our own written connection pooling.

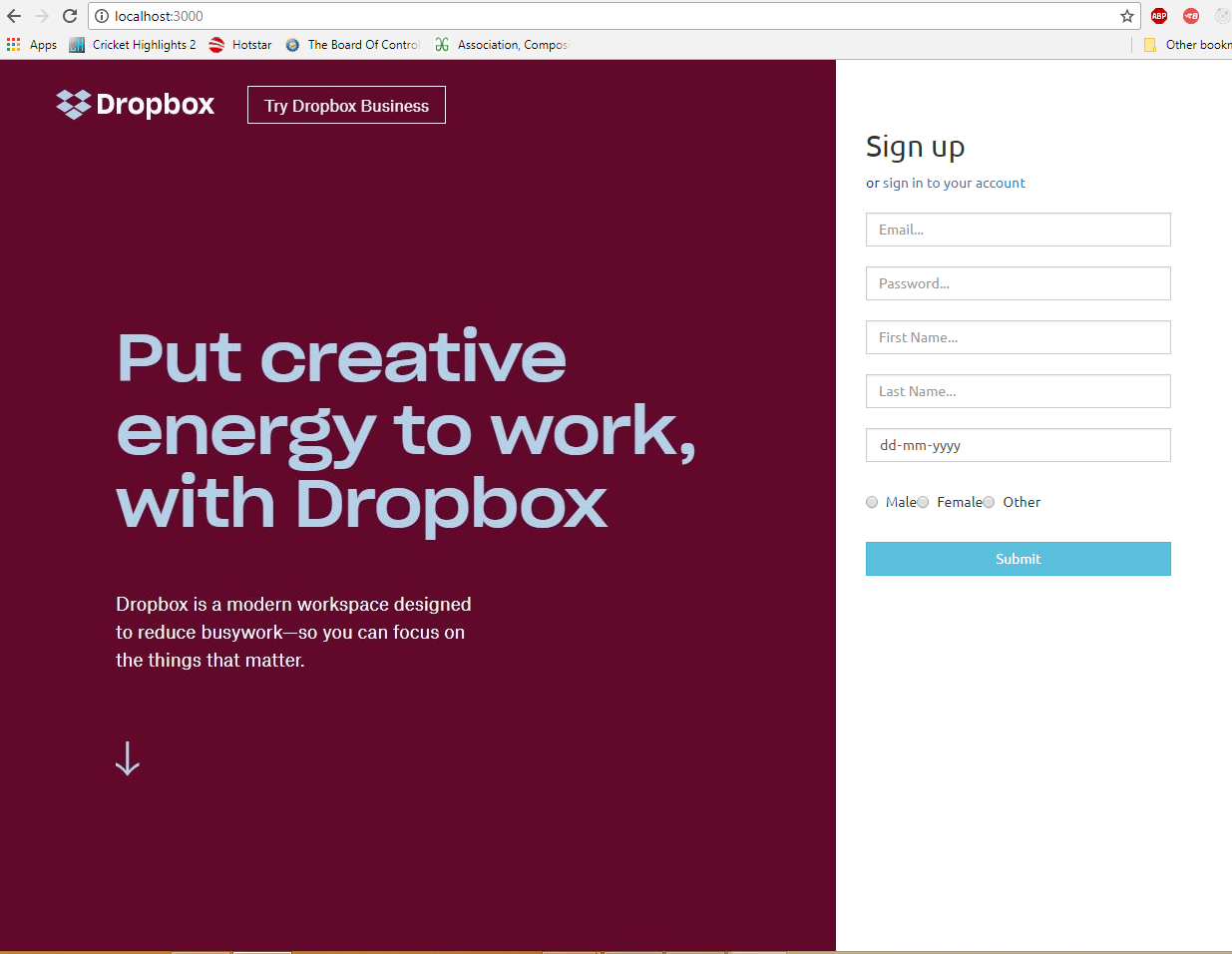
The Front-end is designed with React-Redux, HTML5 and BootStrap 3.7, while backend features Node JS as the Backend Server, Express as Scripting Language in JS and MYSQL Database implemented with connection pooling.

The functionalities include

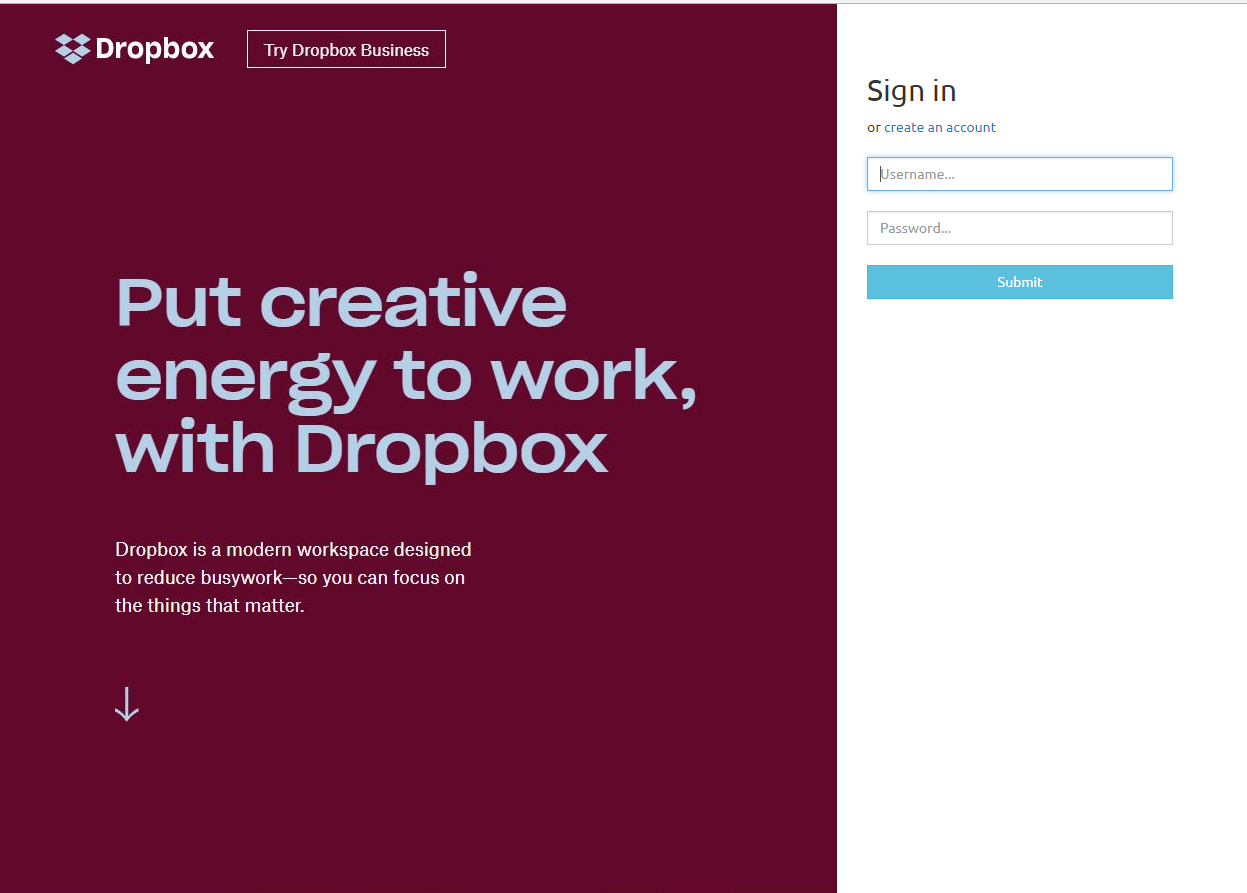
* Sign Up New User
* Sign In
* Logout
* Upload a File(Nested)
* Creating Directory (Nested)
* Deleting a file/Directory
* Star a Folder/Directory
* Un-star a Folder/Directory
* Creating/Deleting a Group
* Sharing the files and folder by Email address
* Sharing with group with which you are part of or Admin of
* Adding Profile
* After Clicking group, Member of the group are shown along with group shared files
* Admin of the Group can add members and delete members.
* Members of the group can view each other profiles if they have submitted
* Activity Report list the files which have been created earlier and then deleted.
* Members in the group are assigned permission to view nested Files and folder shared of which Parent is shared by the members
* Connection Pooling Implemented on Server Side.
* Password has been protected with encryption algorithm.
* Shared Section – show the files which other users have shared with him
* Profile Section – Creating a Profile for the user which can be seen in groups by clicking group members.

*Screen Image Captures:*

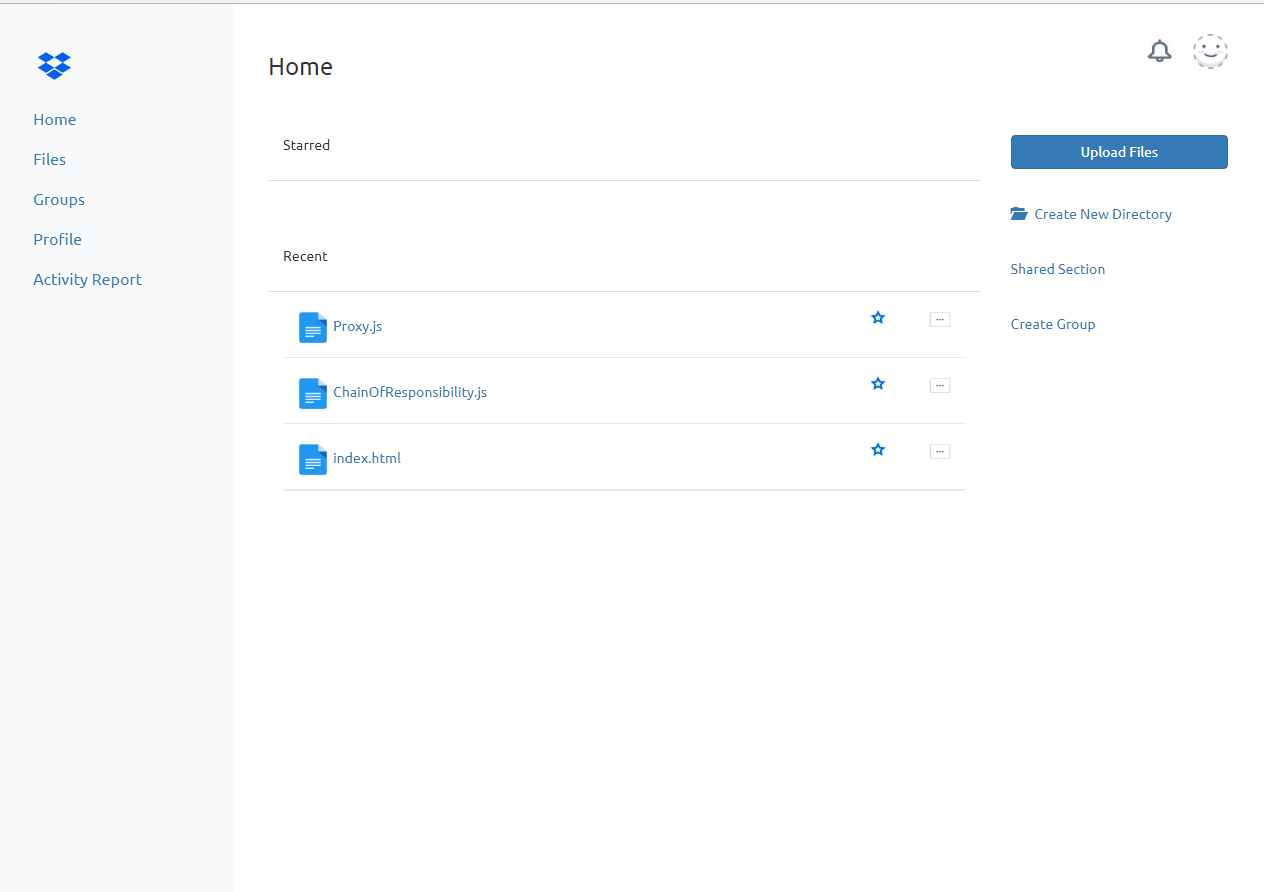
**Landing Page**



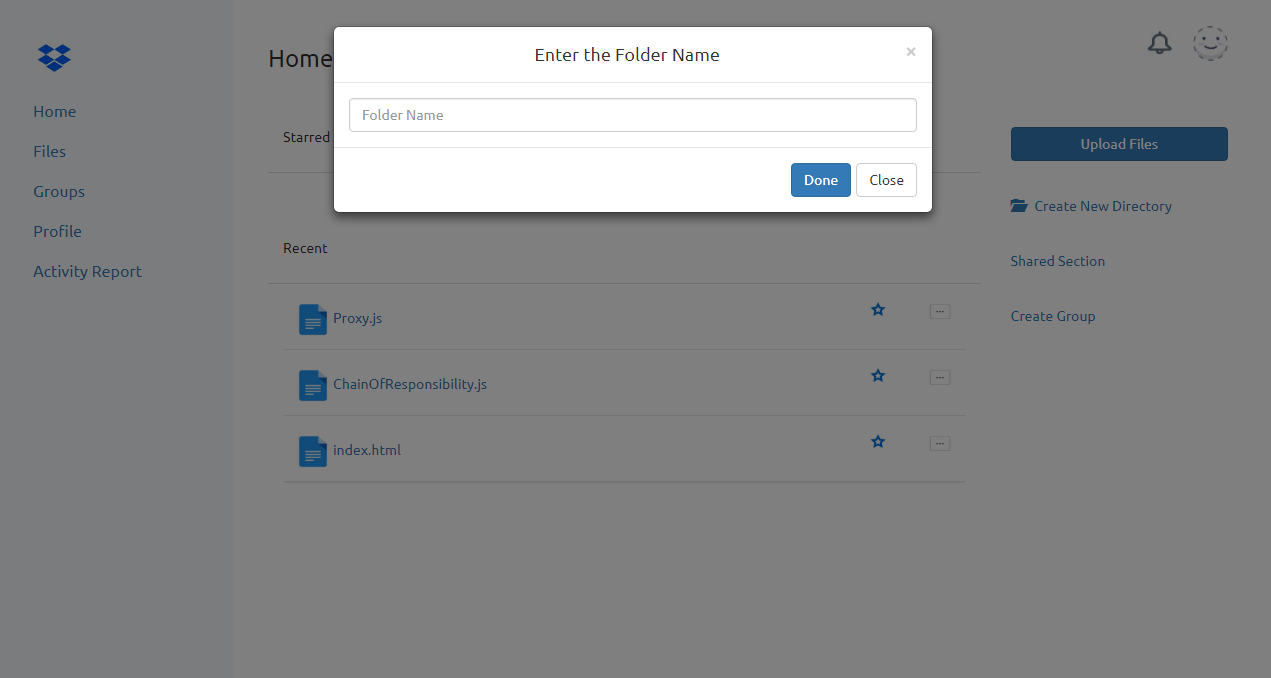
**Login Page**



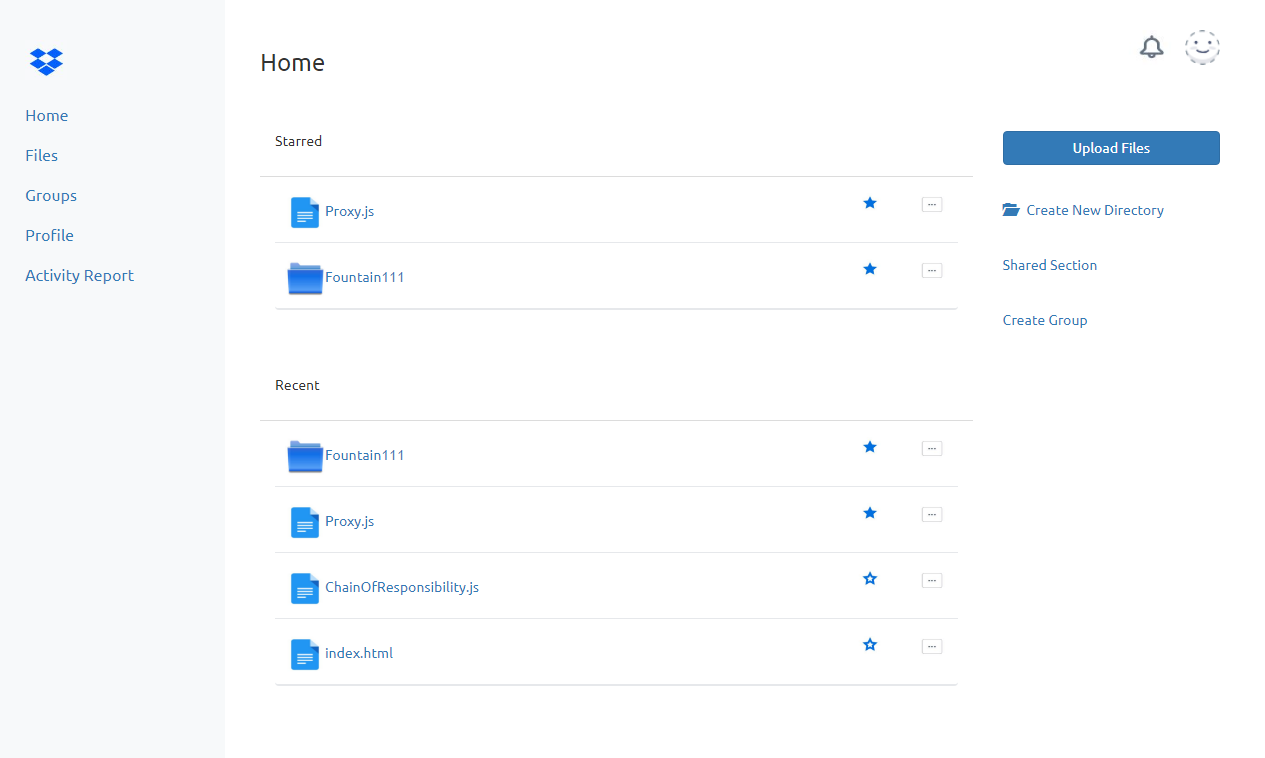
**Home Page**



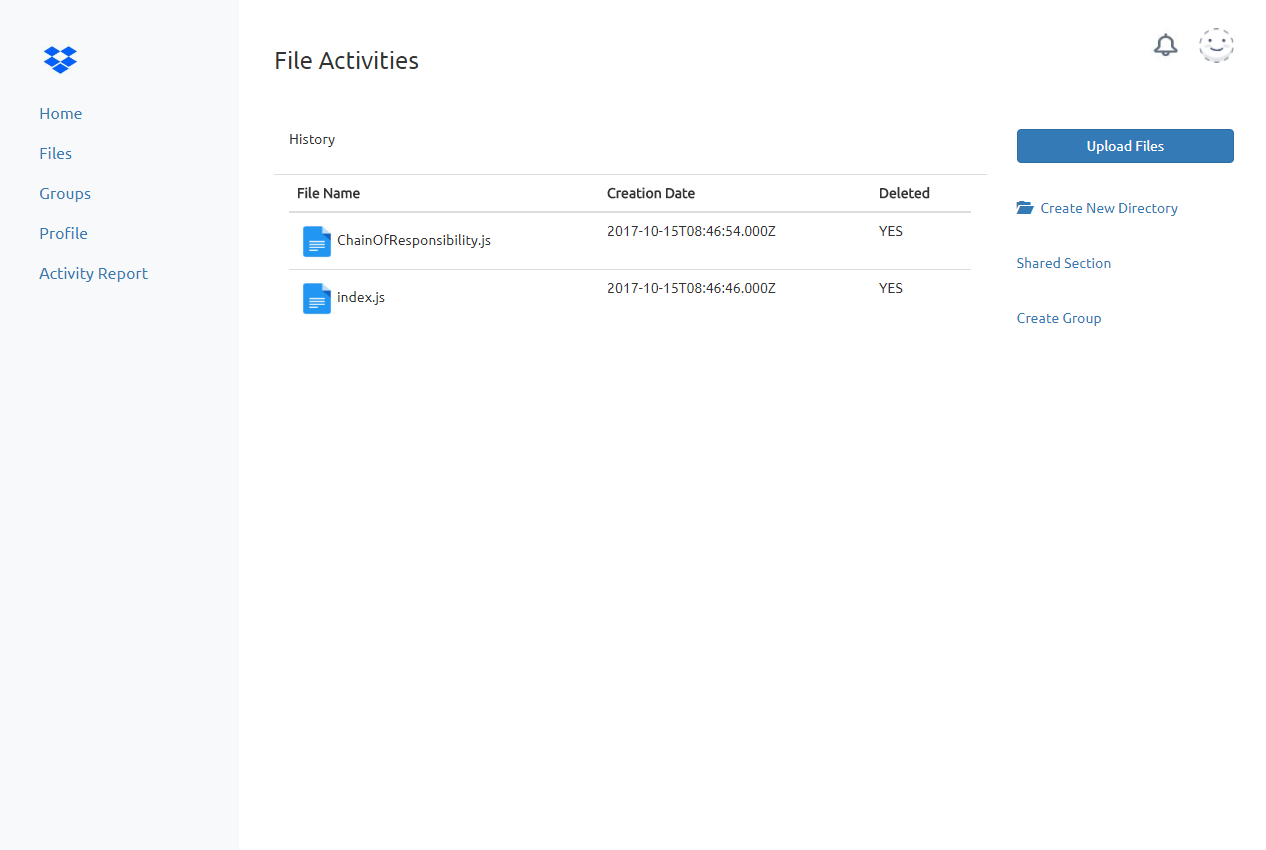
**Creating New Directory**



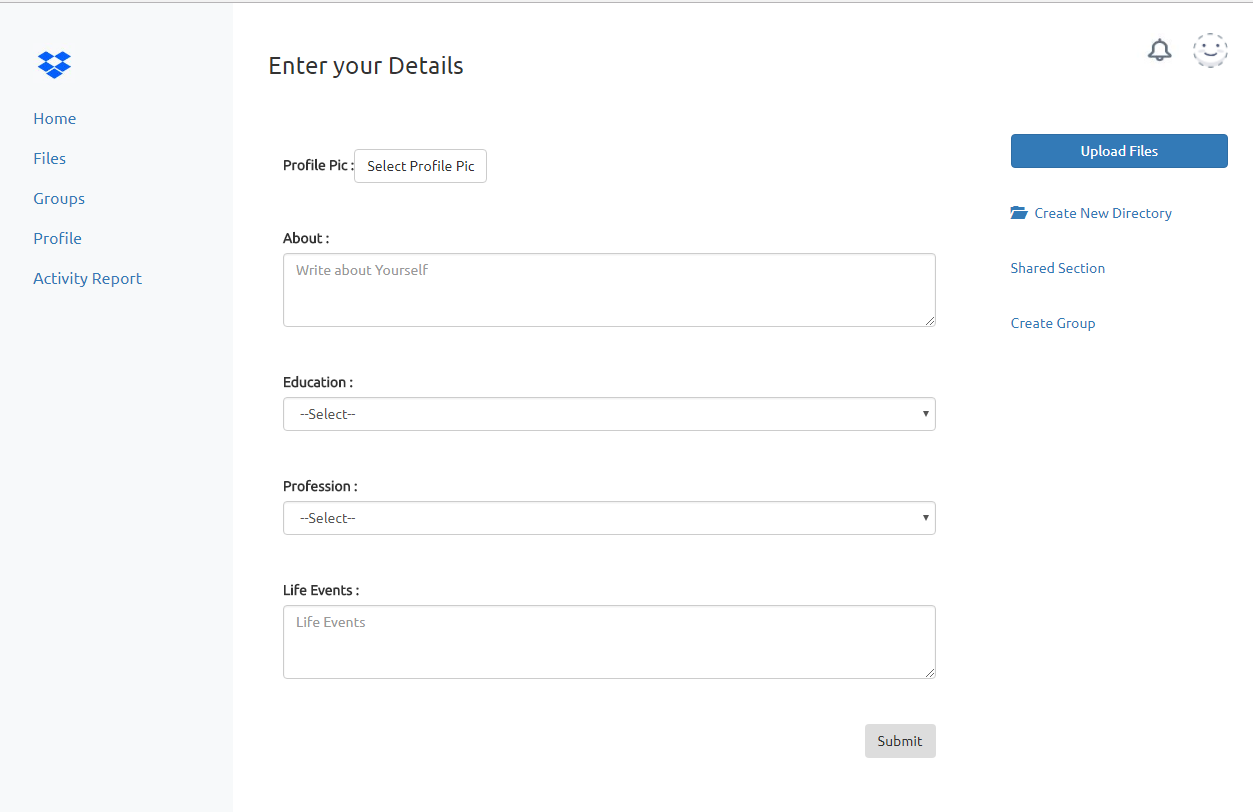
**Starred and Recent Files Section on the Home Page**



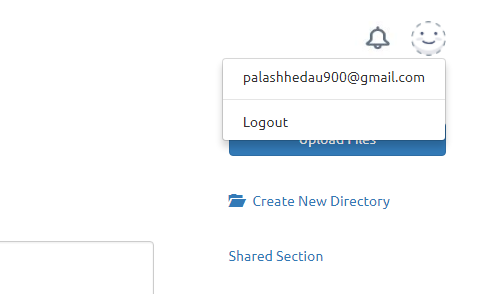
**File Activity Report Section**



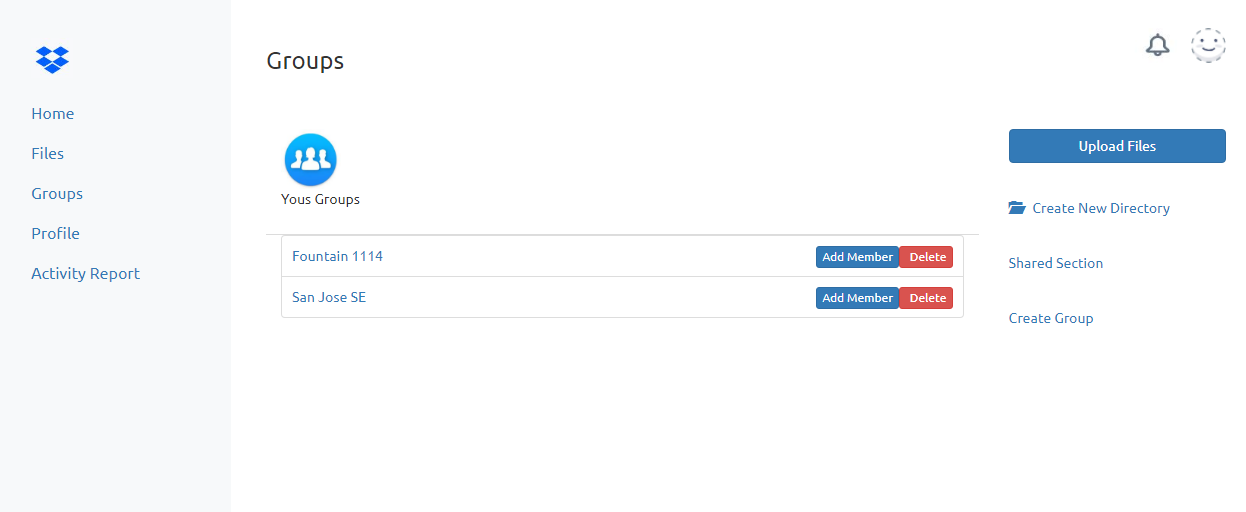
**Profile Page**



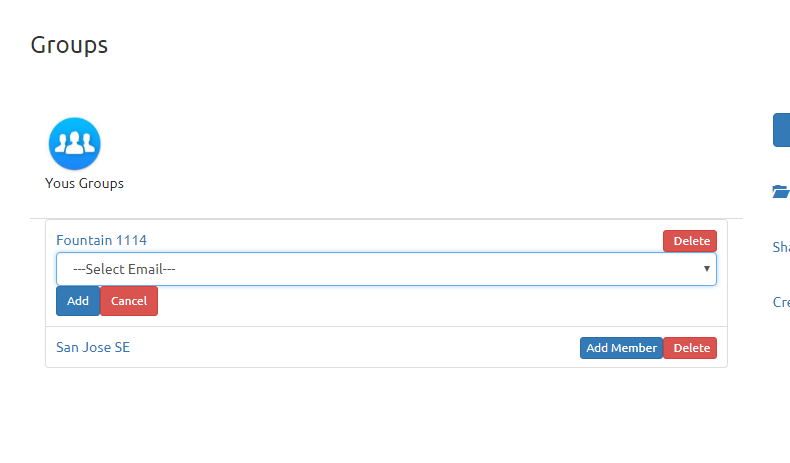
**Logout**



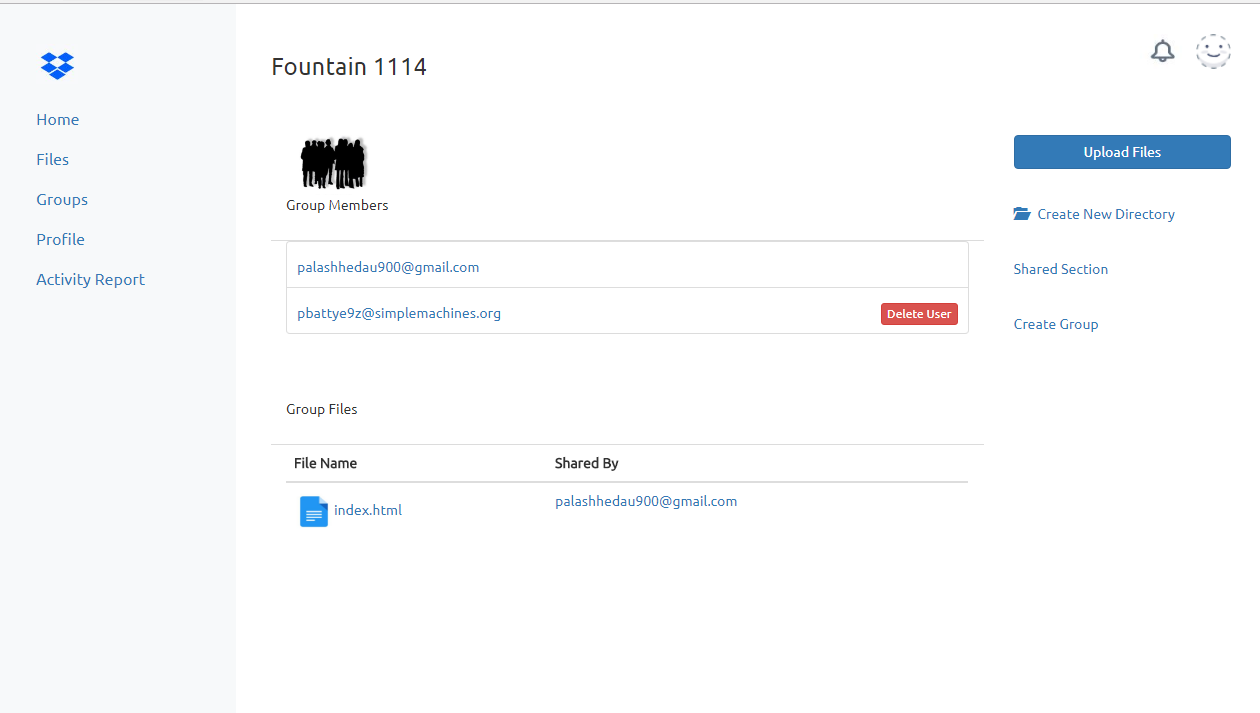
**Group Section**



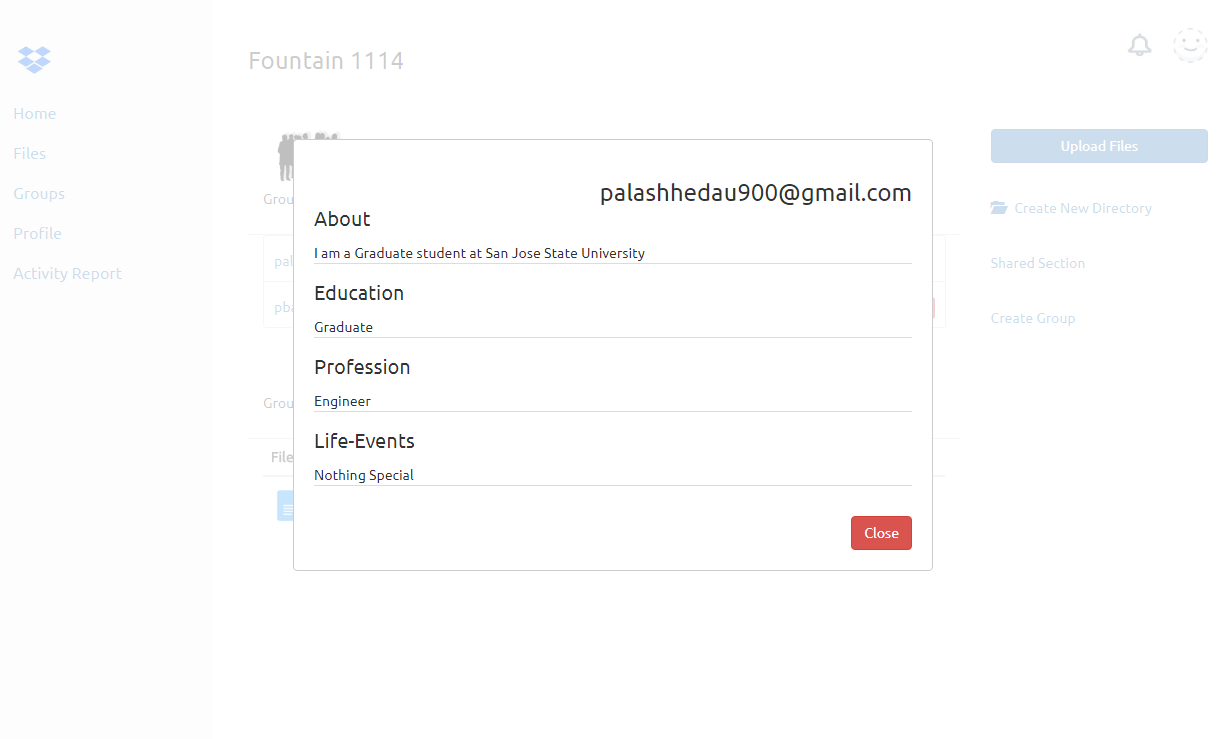
**Adding Members to the Group**



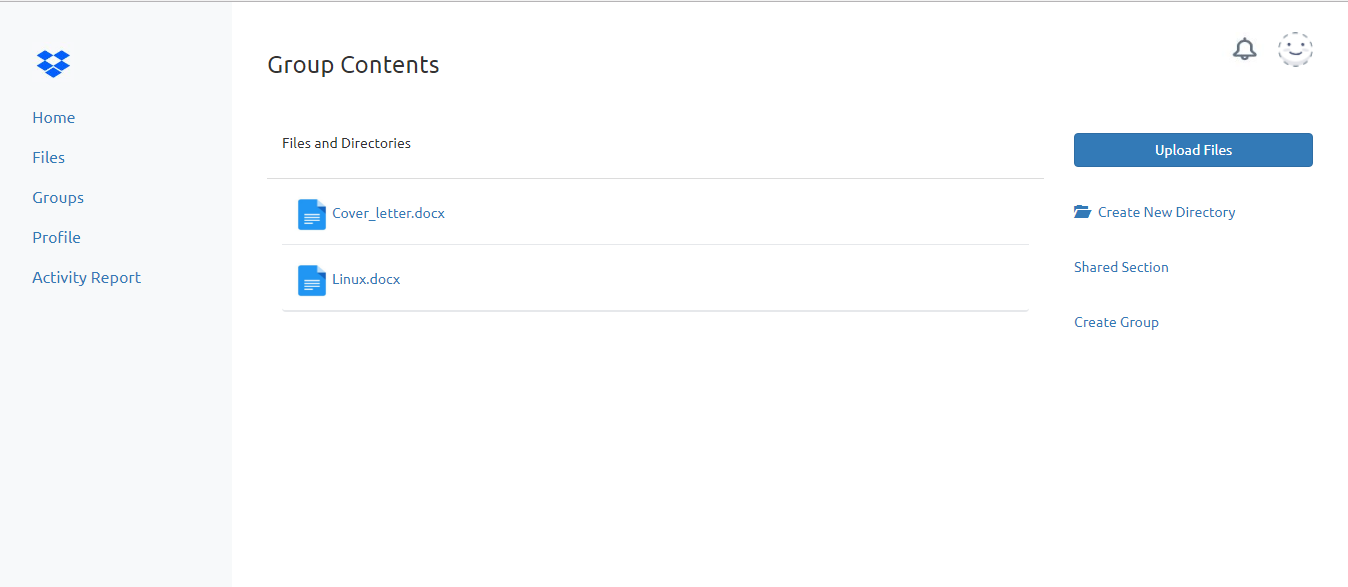
**Group Contents**



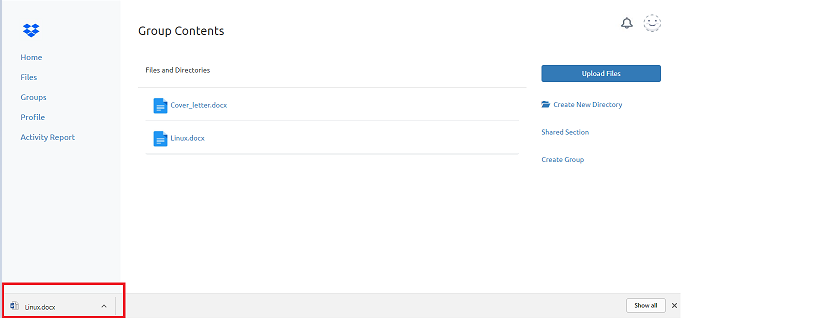
**View profile by clicking on the member**



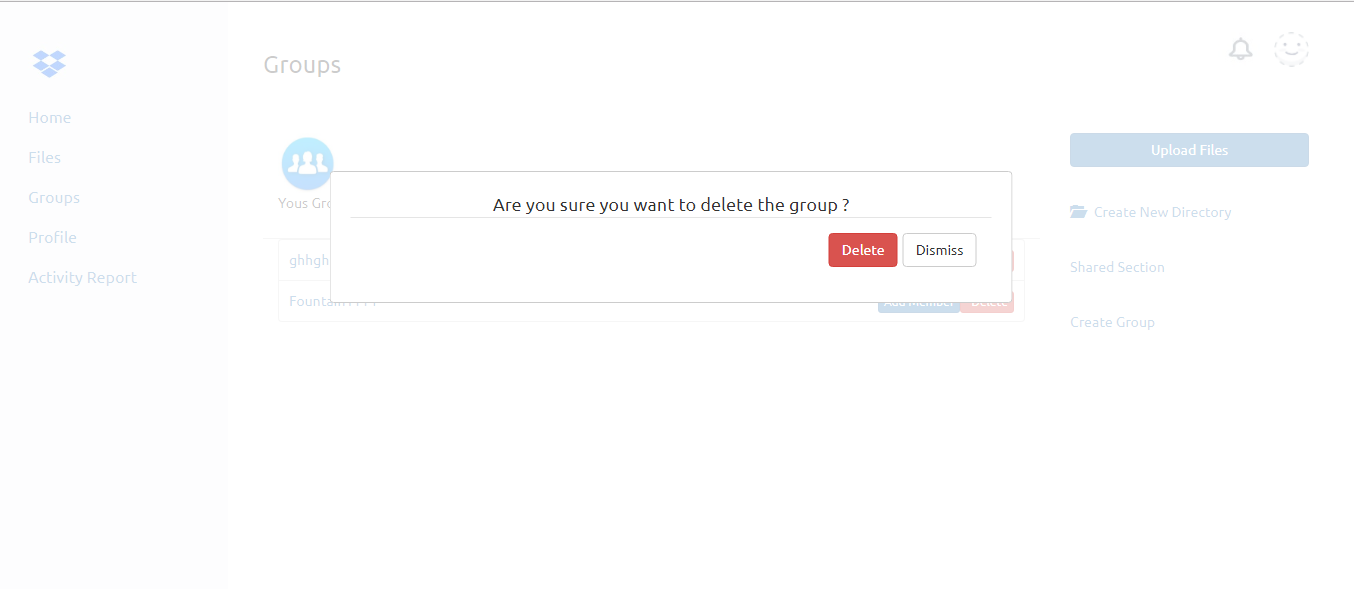
**One can navigate through Files section of the group which was shown below on the Groups Page.**



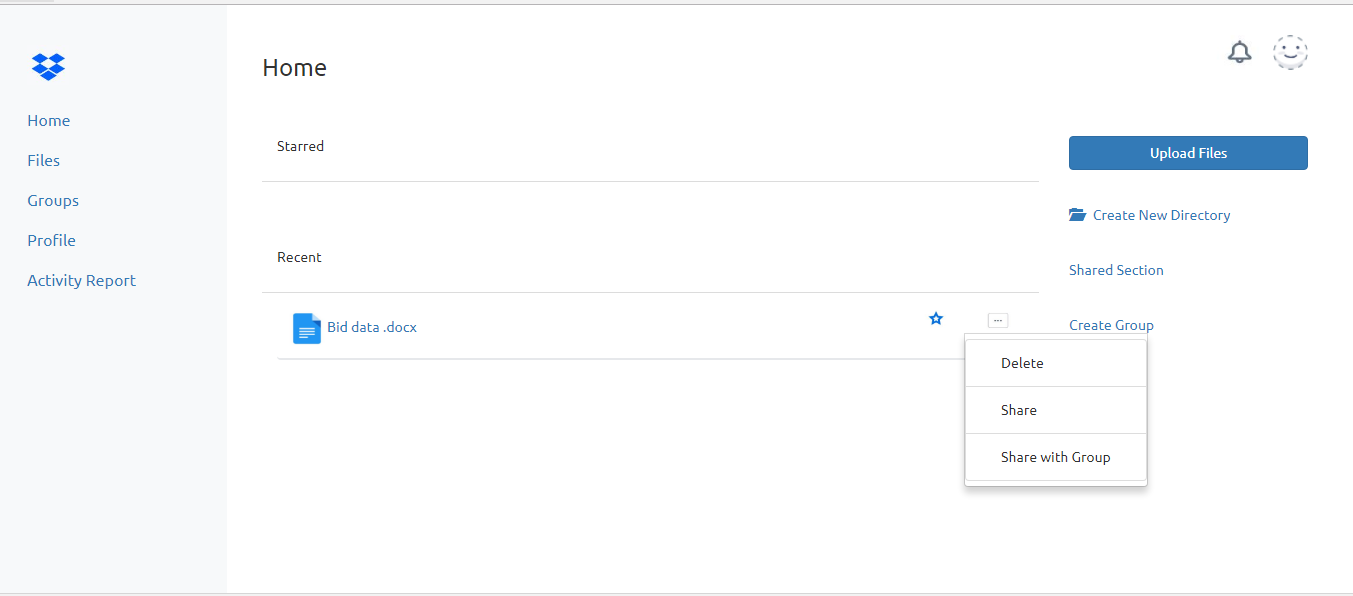
**Download File**



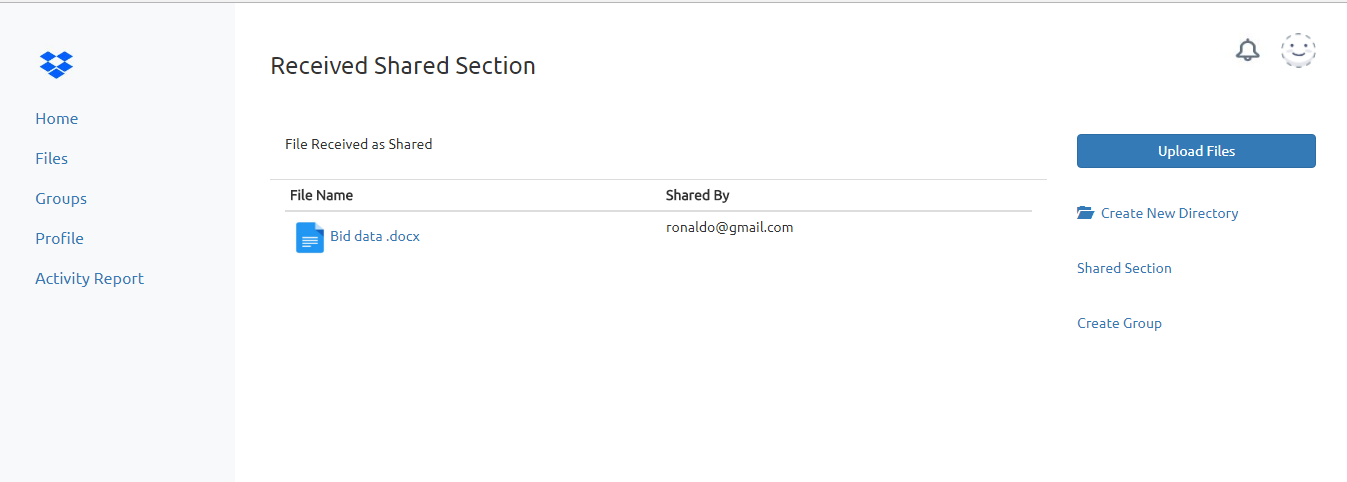
**Delete Group**



**Share File with individual**



**Viewing file shared by others**



*Connection Pooling*

Connection pooling is used for getting a Database connection and re-using it.

I have created my own connection pooling with the help of an array object. The 100 Database connection is created as soon as the server starts and whenever new user comes and ask for Database connection, a DB connection is popped out of an array and uses it. As soon as Database connection is used and no more required, the connection is given back to the array object which is made available to the other if users who are waiting.

If all the 100 connections are consumed and there is some more request then it waits for some connection to be release from the 100 connections, as soon as 1 is available, it grabs it and process the data.

*Sending Large Files with Kafka*

Kafka was allowing files only with maximum 250kb size, so changing the configuration still not worked. I have created the chunks for the file of maximum 100Kb will be sent across the Kafka with every request and but not wait for the data response from the Kafka node server until the last chunk of the file is sent to the Kafka node server. If there is a file of 5MB, 50 chunks of 100KB each is created and with the loop for array, each chunk in the set 50 is sent with the same correlation id, the chunk obtained on Kafka-node-server accepts the chunks till the last chunk is obtained.

The chunks are stored in the object which waits for last chunks to arrive and as soon as last chunk is obtained, the files is combined and stored in Database.

Similar approach is taken while downloading the files.

Max size tested with above approach is 50MB.

*Questions*

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

* The Authentication process used in Lab1 was Bcrypt and now in Lab2 we are using local passport authentication.

In Bcrypt, we had to provide our own salt Round but now in Passport Encryption, the salt round is generated randomly for different users of the application.

If 2 users have submitted same password and we are using same salt for application, it can create conflict.

With using Passport authentication, it can be avoided.

Passport can be combined with lots of other strategies such as Facebook, Google and is also very lightweight.

Passport can also be used to store sessions along with other session strategies in Database and on logout it can delete sessions automatically.

Also Passport store whatever thing required in session and can retrieve it.

Whereas the Brcypt doesn’t provide this number of functionalities. It just hashes the password and then we compare it with the one we have provided for login.

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

* Application with Kafka messaging Queue took more time than compared to application without Kafka.

The time required to pass the object from node server to Kafka and then receiving it by Node-Kafka server, processing it, giving back the result takes some time.

Also the Queuing of the request is also responsible.

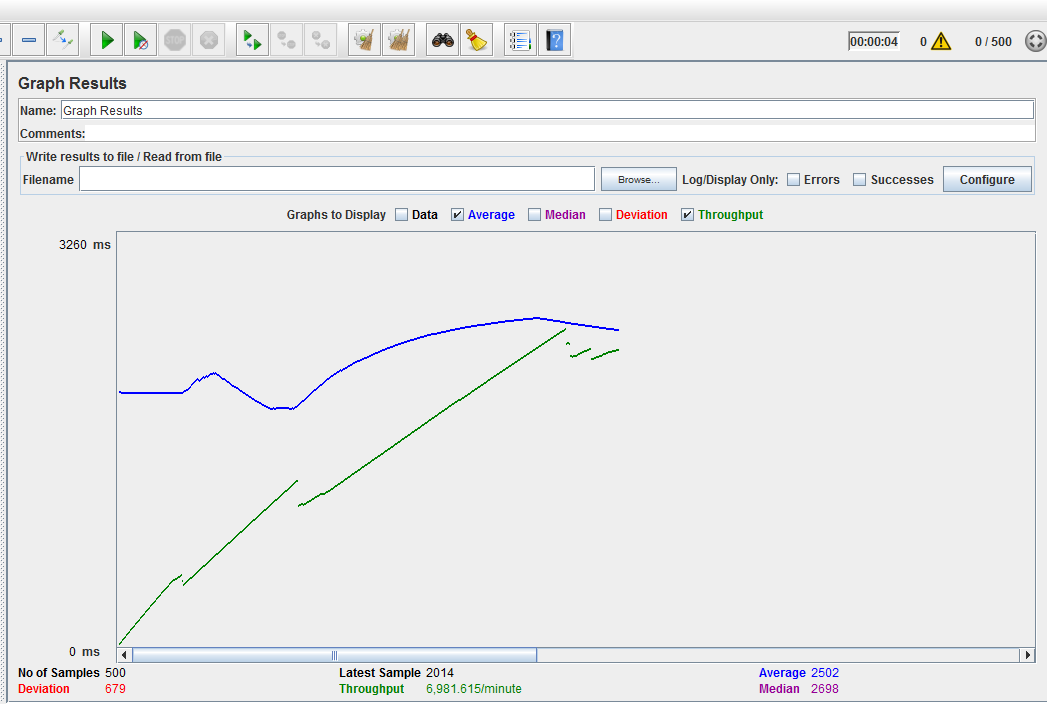
But addition, Kafka makes the application more secure and durable with very minimum data loss. The Durability is high with high throughput.

Also the application becomes fault-tolerant, so adding it makes your application more secure and contribute the application to become more distributed.

Also Kafka can hold events longer, so even if there is some exception on the Kafka-node server, the messaging queue can hold onto events and later can be dispatched when server is up.

Analysis without Kafka with “getAllUser” api with 500 calls.

With Kafka, it is shown below in Jmeter Testing section.



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.**

* Currently as per my system architecture, the way the conversion has gone pass by from MySQL to MongoDB, I would prefer to store most of the data in the MongoDB. MySQL is fast but not as compared with MongoDB.

Sometimes with MySQL, we have to create lots of joins to accumulate data for the application.

Furthermore there are chances for the duplicate data to arrive if not programmed properly.

With MongoDB we can store data in arrays inside arrays inside objects, we can have N level of hierarchical data inside a single document.

We have tables pre-defined with columns in MySql and does not accumulate more data than that, but with MongoDB, we can have flexible columns or data inside the document.

So, all the data related to user, I will store in the MySQL and then data related to groups and files, will be stored in MongoDB.

Also the data insertion and retrieval becomes faster when compared to MySQL, so if files needs to be maintained in DB, then definitely Mongo comes into picture.

So for light/sensitive data, I would prefer MySQL, for heavy/large I would prefer MongoDB.

*Performance*

*Jmeter Testing for 100,200,300,400,500 connection*

* It was observed that our own created connection pooling had better performance than that of DB provided connection pooling. It took slightly less time compared to the DB provided one.

On the other hand the time taken by the application without creating pool was the lowest. It took slightly more time compared to that of DB provided pooling.

So, it is advisable to use connection pooling created by you, if not then use DB provided connection pooling as the second choice and creating connection every time when API is called is not preferred. You should avoid such situation in your application.

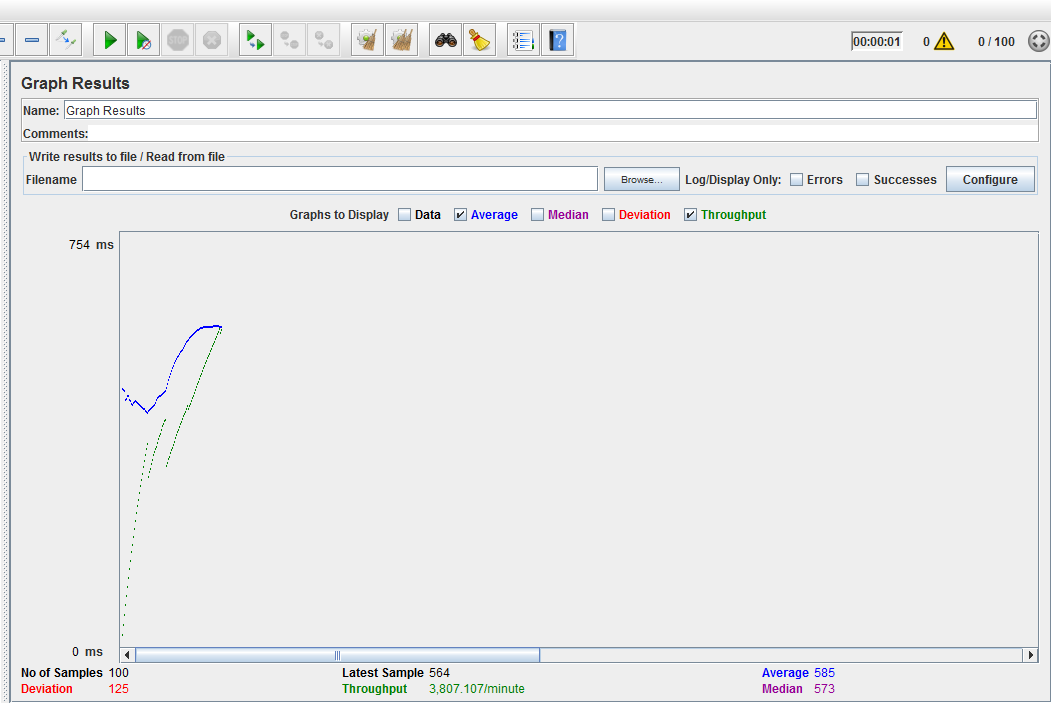
Preference Sequence by the below test

1. Own Connection Pooling
2. DB provided Connection Pooling
3. No Connection Pooling

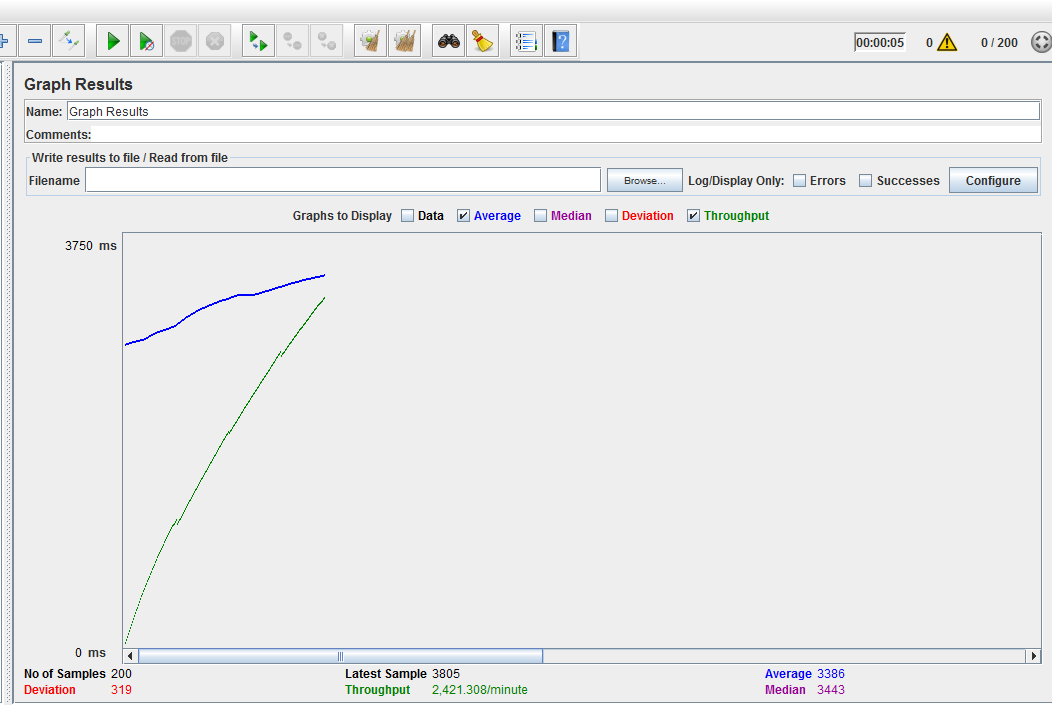
Below Diagrams provide analysis for the same which proves above statement.

Own connection pooling

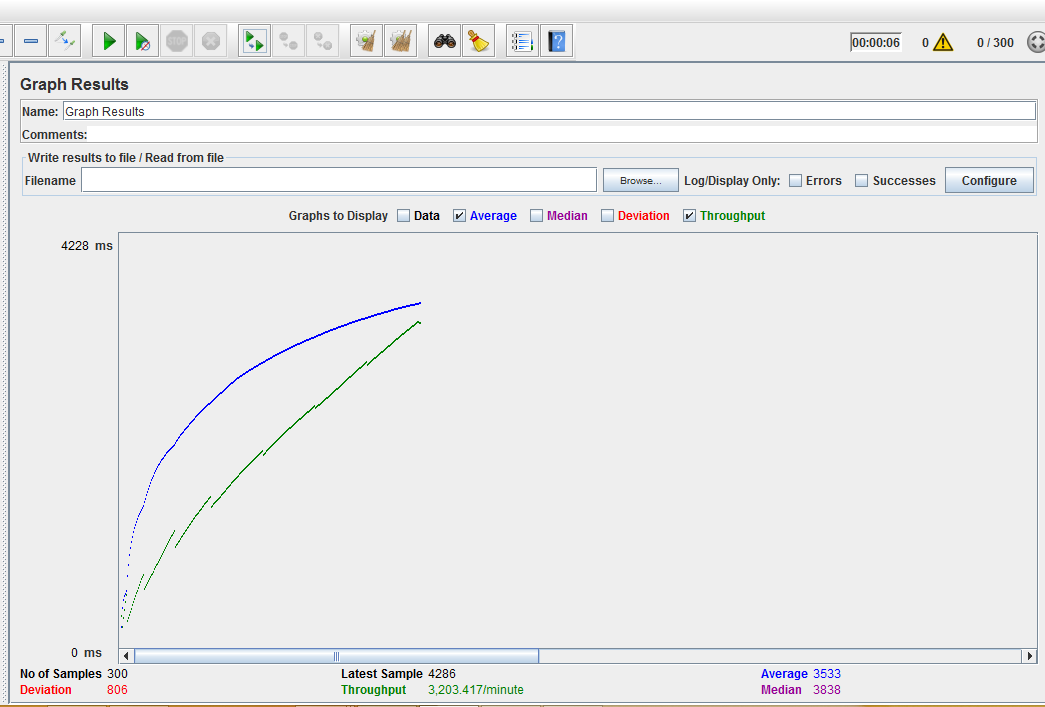
100



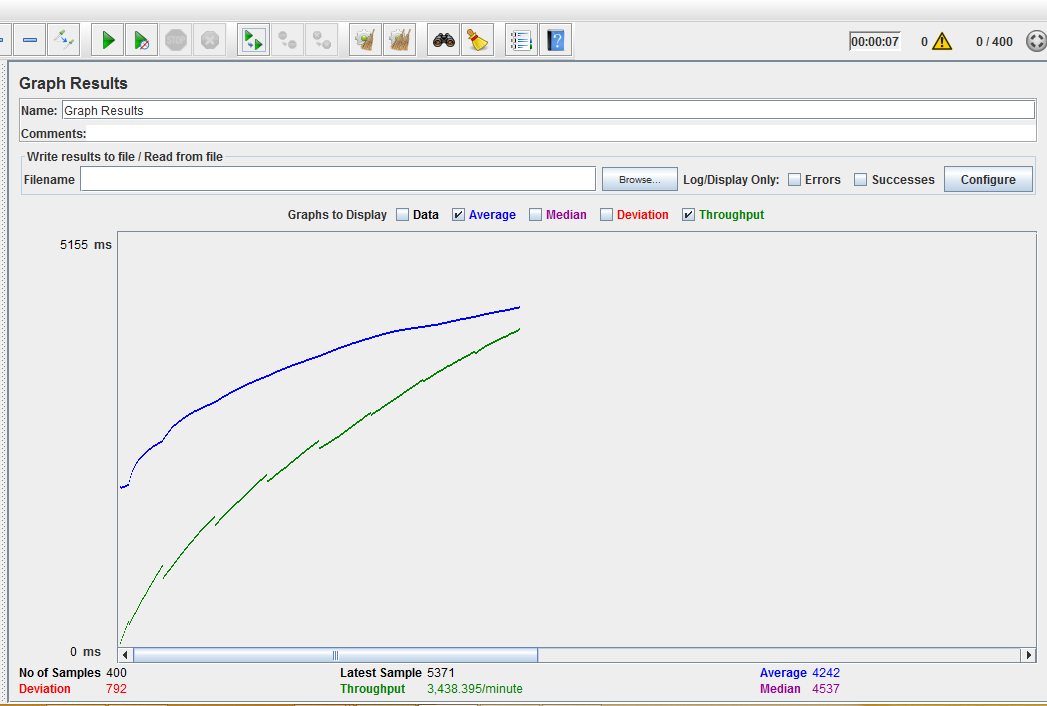
200



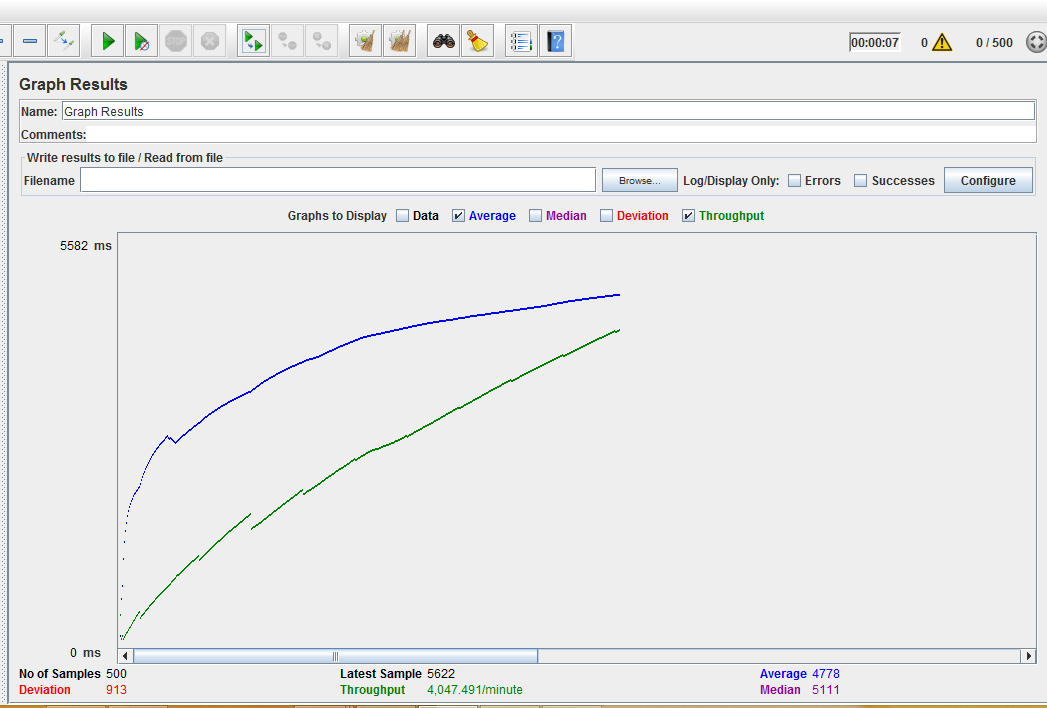
300



400

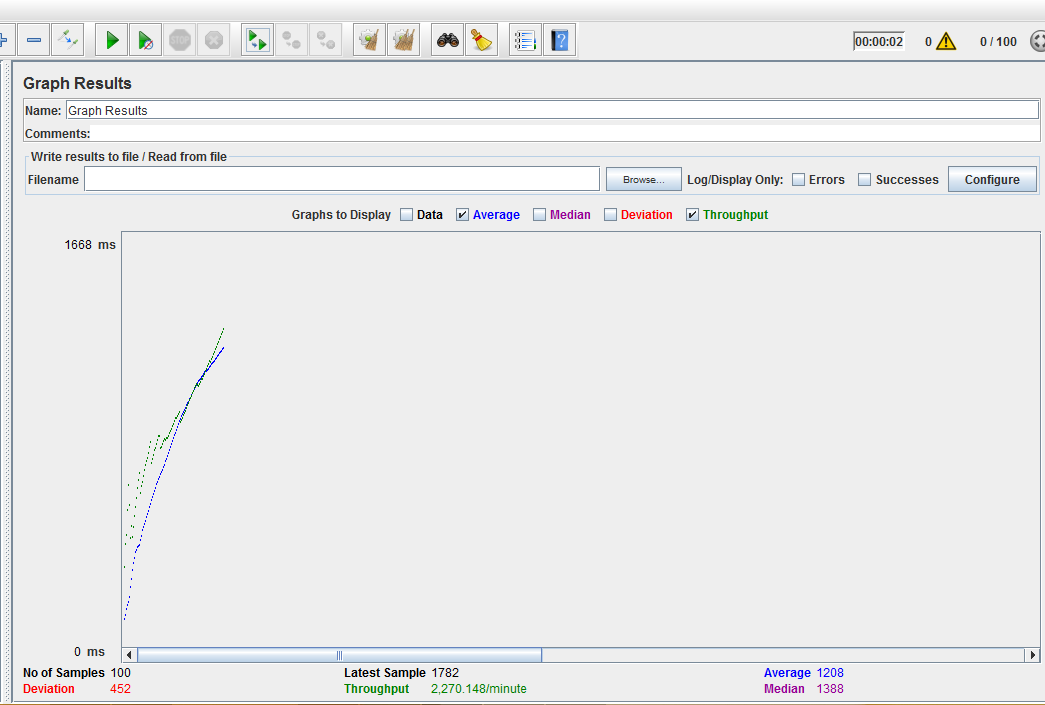


500

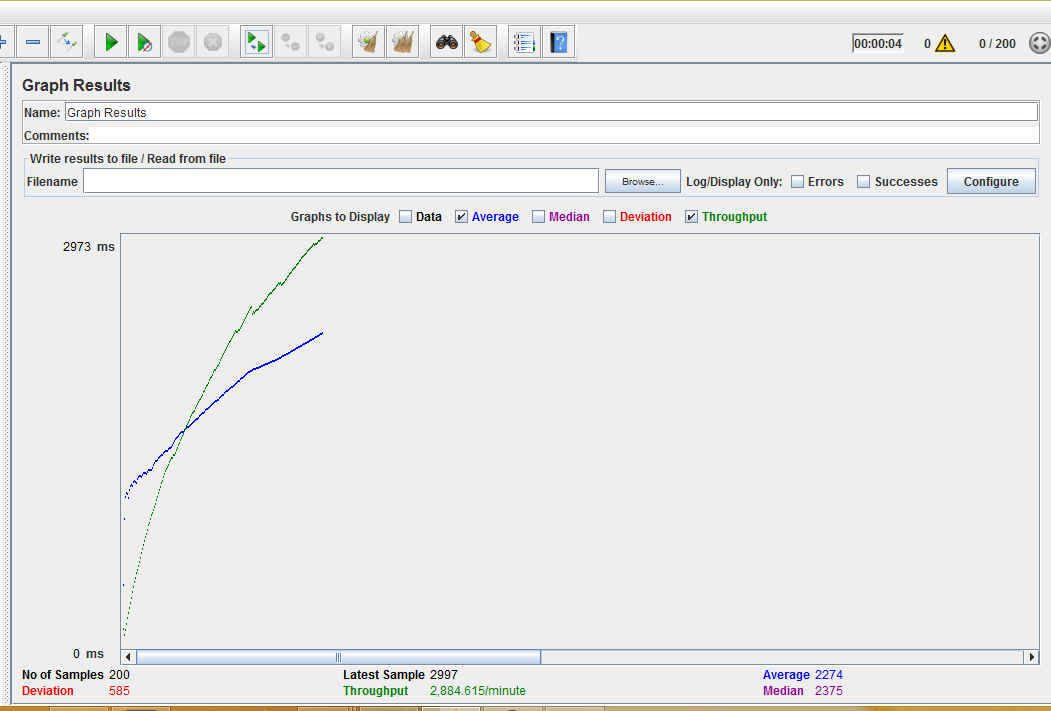


DB Defined Connection Pooling

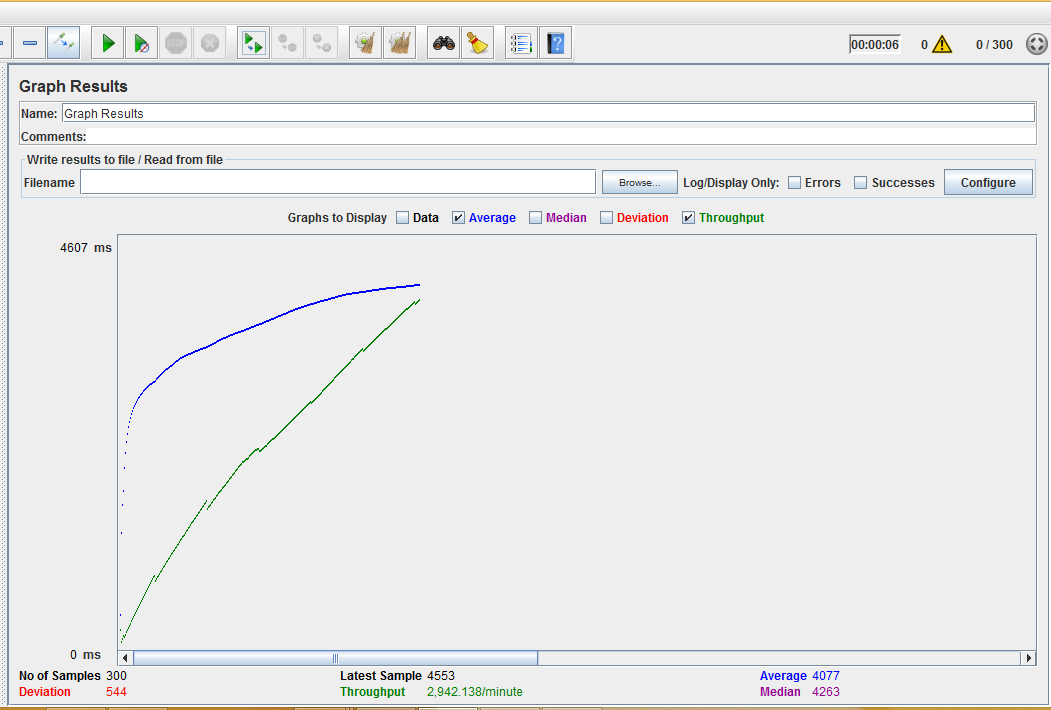
100



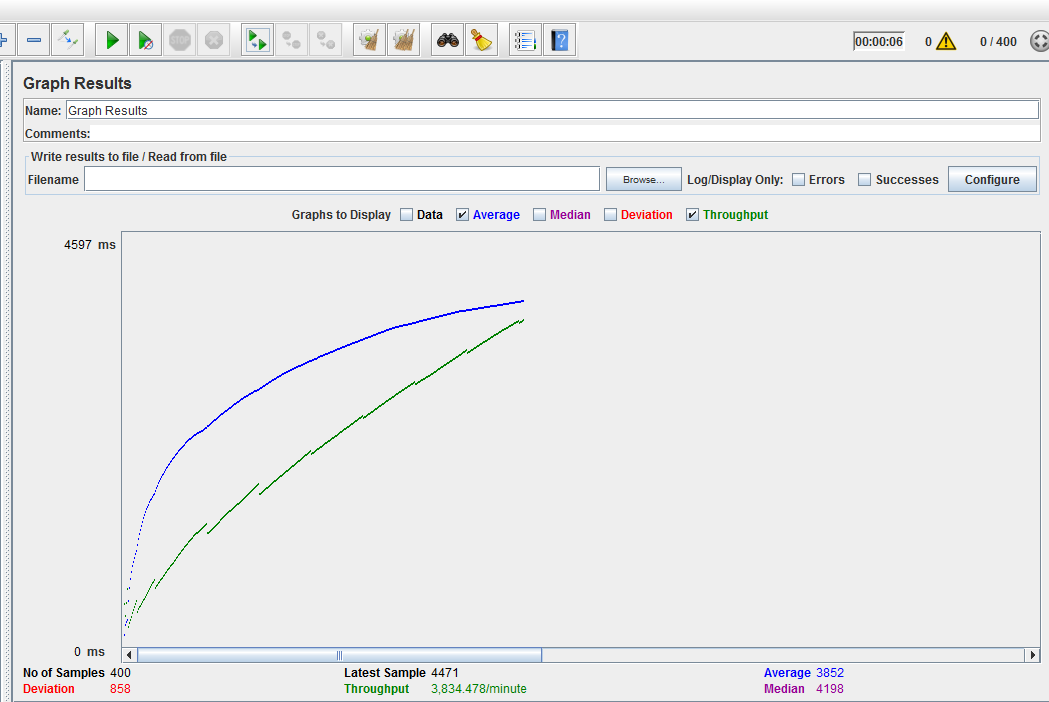
200



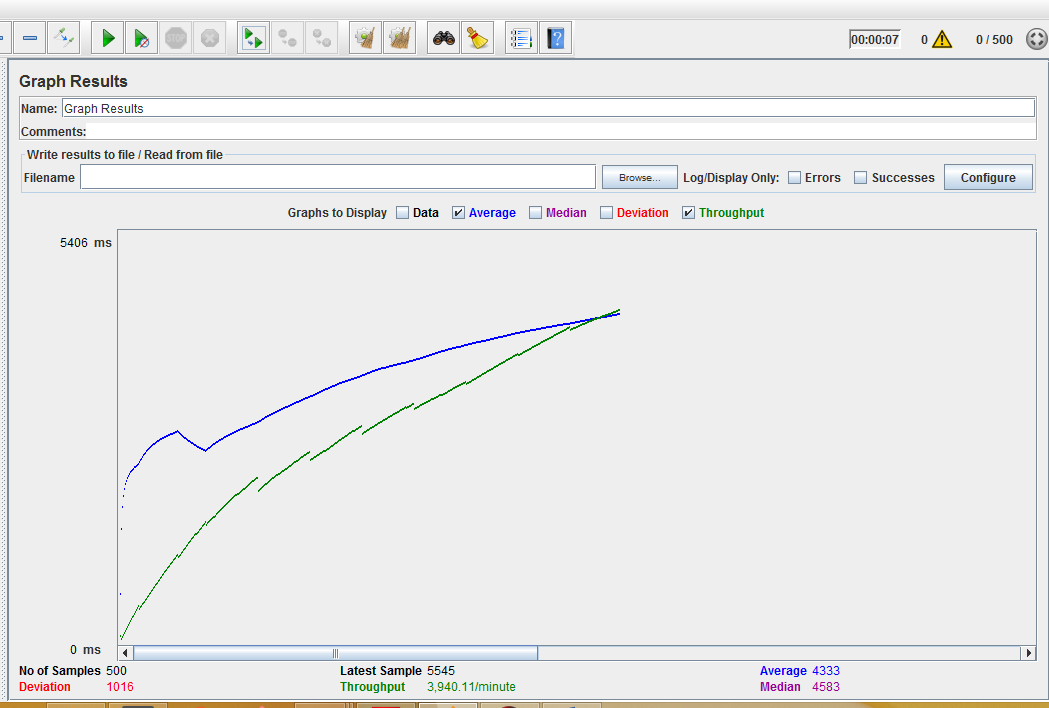
300



400

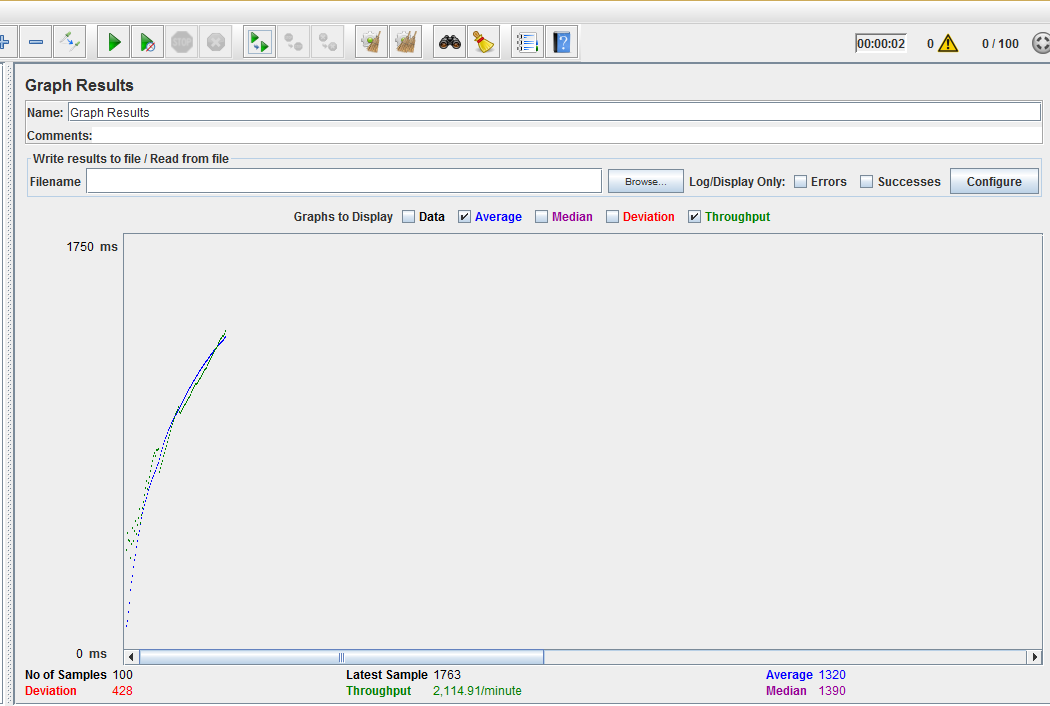


500

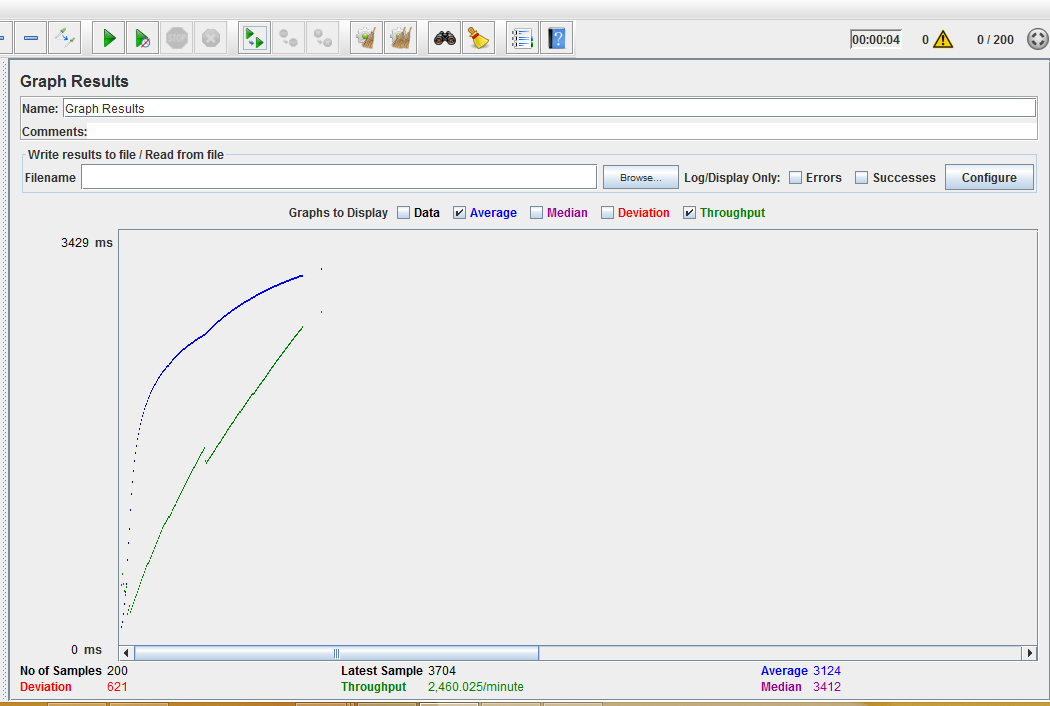


No Connection Pooling

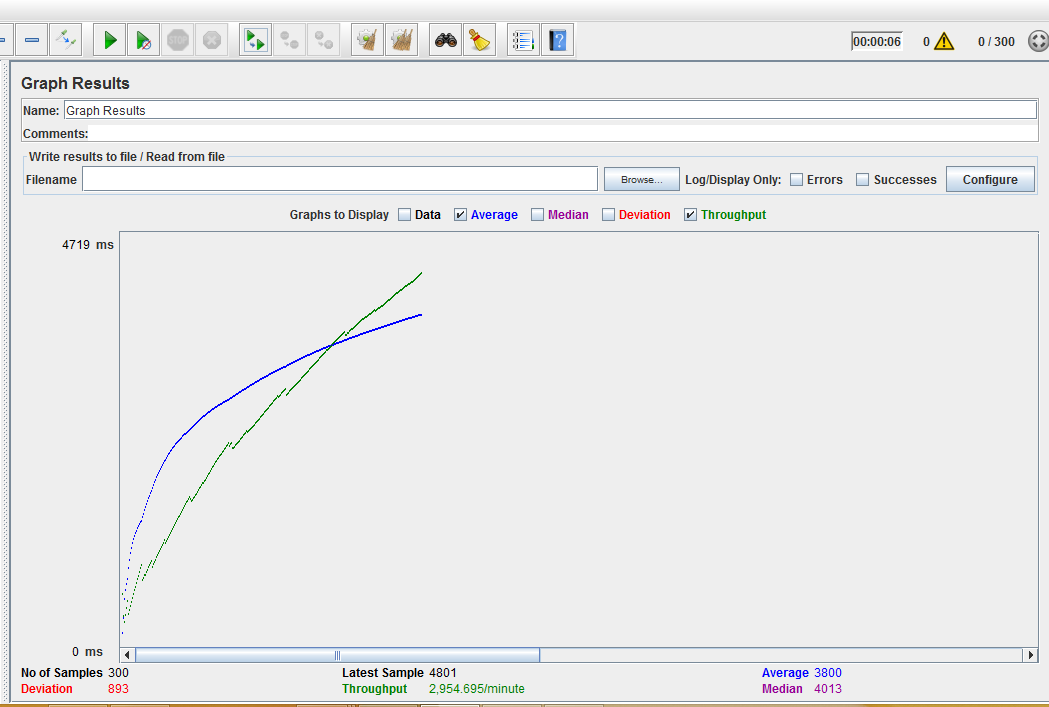
100



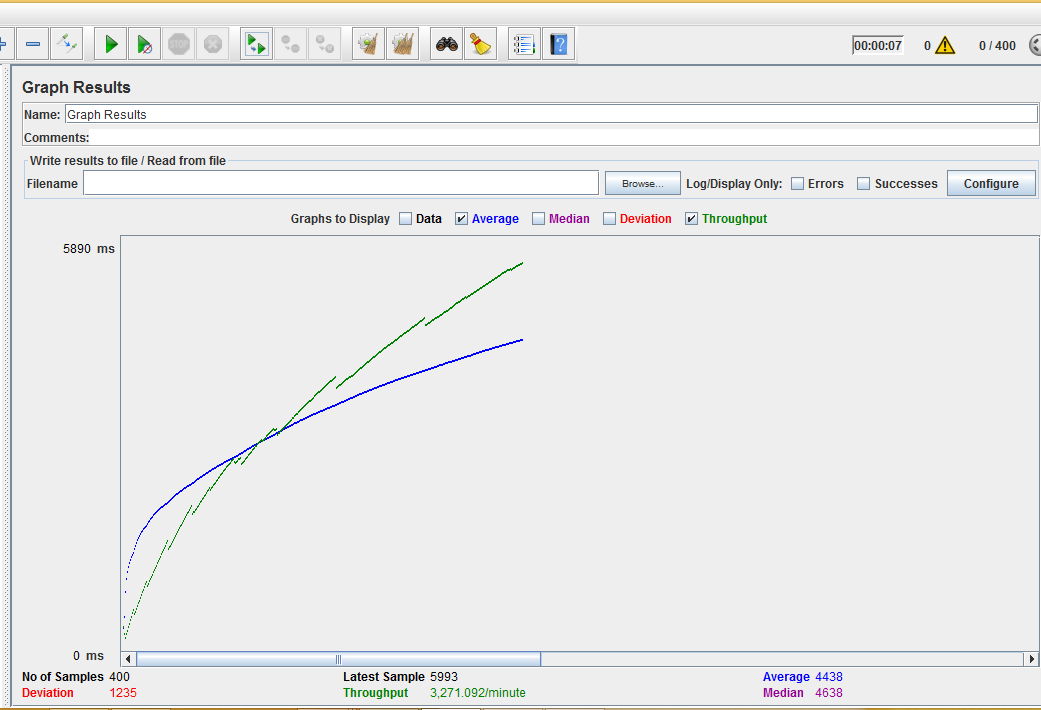
200



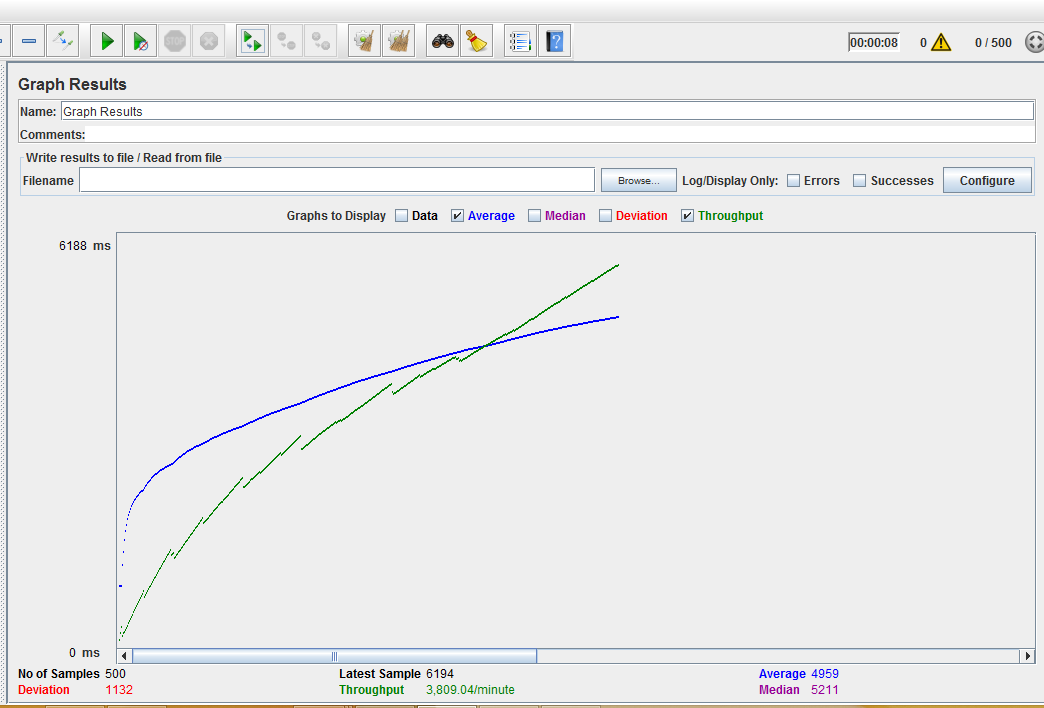
300



400

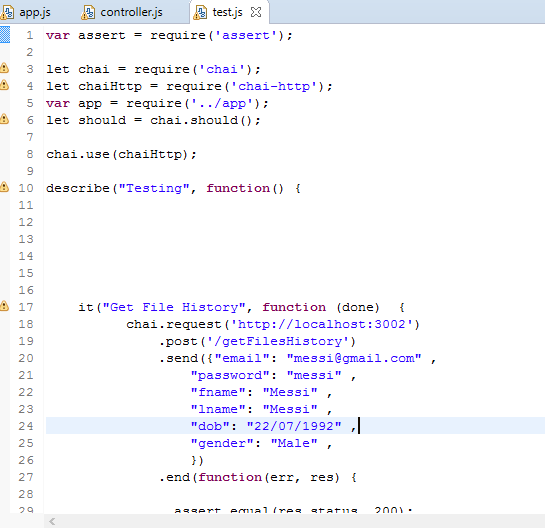


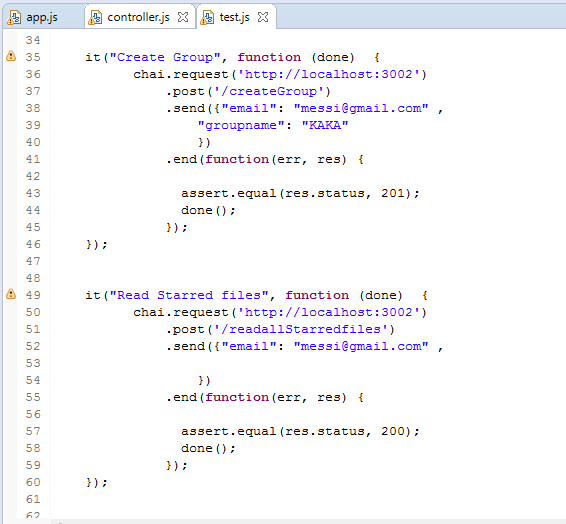
500



*MOCHA Test-Cases*

Code for some of the test cases







Result of the Mocha Test-cases (10/10 Passed)

