**CMPE 273 Lab 2 Assignment**

by Sreedeep Katragadda (011545222)

**GITHUB URL:**  <https://github.com/sreedeepk/CMPE273>

**Introduction:**

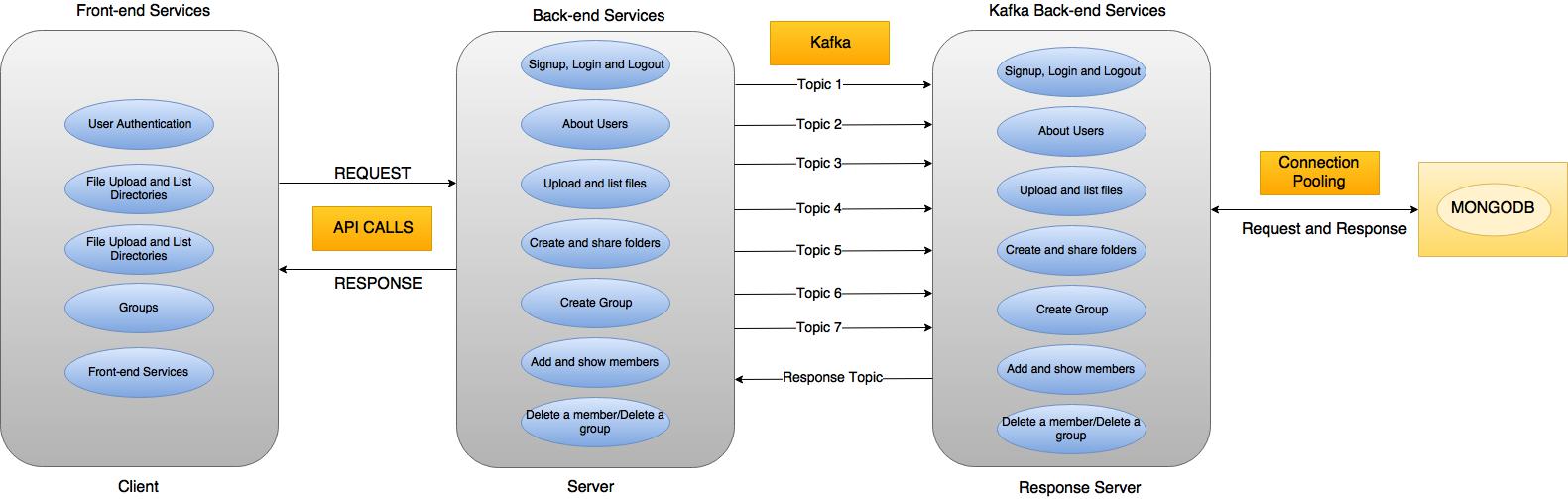
The purpose of this lab is to design and implement distributed service oriented application using **Kafka**. In this lab, we will create a client-server pair where client and server will communicate through **Kafka Streams**. For database, we will be using **MongoDB** and our sessions are stored in it. For user authentication, we are using **Passport**. On the server side, we should develop a **“Prototype of the Dropbox application”** which should perform the following tasks:

* **User functionalities:**
  + Sign up a new user.
  + Sign in existing user.
  + Sign out.
  + Upload a file.
  + List a file.
  + Create a directory.
  + Star a folder/directory.
  + Share a folder/directory by email/name/link.
* **Users account:**
  + About: User overview, Work and Education.
  + Interests.
* **Providing file list and activity report.**
* **Group functionalities:**
  + Create a group.
  + Add member in a group.
  + Show members in a group.
  + Assign access permission to a directory.
  + Delete member from a group.
  + Delete a group.
* **Should perform connection pooling for database access.**

Here, we are using our own connection pooling for database access. Testing for this Dropbox application is done using **JMeter** and **Mocha**. We then create a client side to include all the functionalities implemented by the web services. The client and server will communicate through **Kafka**.

**System Design:**

**Architecture Diagram:**



The above image is the architectural design of our application.

As we can see, there are two servers on the back-end. The server directly communicates with the client using regular API calls. The job of this server is to receive requests from the client and pass it to the **Kafka Back-end Services** via **Kafka**. As we can see the requests from the client are received and pushed into the topic by this server, then the Response server pulls out these messages and communicates with the **MongoDB** and generates a response. This response is further pushed into the “response\_topic”. Then the messages from the response\_topic are pulled by the server and sent to the “Front-end Services”. This is how our application works and can be easily understood with the help of the architectural diagram.

**Server:**

The server is to design and create a **“Dropbox application”** to demonstrate RESTful services. The server side of our application is divided into two parts as: back-end and kafka-backend. Here, in the back-end lies our access end points and routes to the application. In the kafka-backend we will put our kafka configuration such as topics to the end points and tell the server to communicate to the client side via Kafka.

//About Package.json of server

There are several end-points for the server 2 in the back-end part and they are described as below:

* **POST '/users/signup'**

This is the end-point which is used for a user to sign up. The parameters for this end-point are Email, password, First Name, Last Name, Overview, Work and Interests.

* **POST '/users/signin'**

This is used for the user to sign in to their Dropbox account. The parameters for this end-point are Email and password.

* **POST '/users/signout'**

This end-point is used for the user to sign out of their account.

* **GET '/users/info'**

This end-point can be accessed by using get method. The function of this is to read the info of a certain user.

* **POST '/files/upload'**

The use of this end-point is to upload the files into their account.

* **GET '/files/:emailid/'**

This is used to display all the files.

* **GET '/files/:emailid/:id'**

This is used to display a particular file uploaded by a particular user.

* **POST '/groups/create'**

This is used to create groups in the Dropbox.

* **POST '/groups/:id/members'**

This is used to create members in a particular group. The group can be accessed using the group ID.

* **GET '/groups/:id/members'**

This end-point is used to display all the members in a particular group. This group can be accessed using the group ID.

* **GET ‘/folders/:id/share’**

This end-point is used to generate a link to share the folders in the Dropbox to other users.

* **POST ‘/folders/:id/star’**

This end-point is used to star a folder/directory by the user.

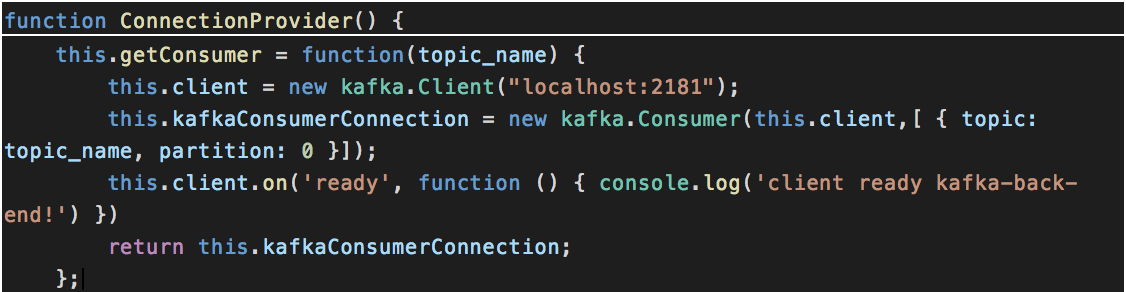
* **DELETE '/groups/:id/members'**

This end-point is used to delete members from a group. The group can be accessed using the group ID.

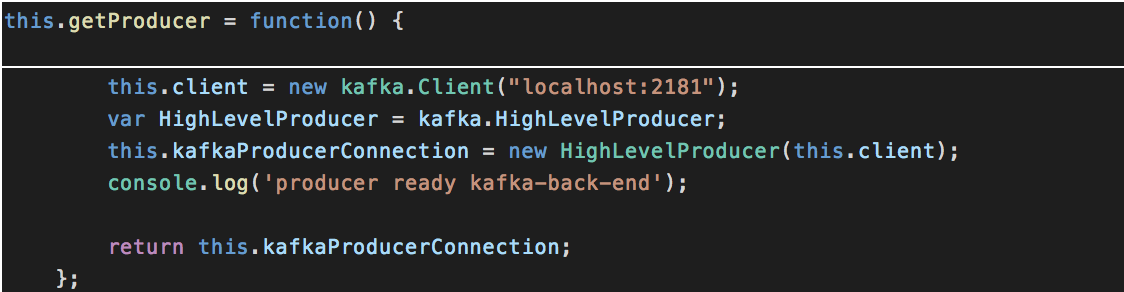
* **DELETE '/groups/:id'**

This end-point is used to delete a group. The group can be accessed using the group ID.

These are several end-points required to run our server side of the application. Now, we will see how these end points from server will pull messages from Kafka topics and how the messages are pushed from the client into these topics. The Kafka Stream connection to our application is placed as below:



We can see from the above code that, the function ConnectionProvider() is used to establish a connection. This above code is used to establish a connection betweem Kafka topics and consumer(back-end). The connection between the producer and Kafka can be established using the code below:



In this way, we can establish a Kafka connection to our producer and consumer.

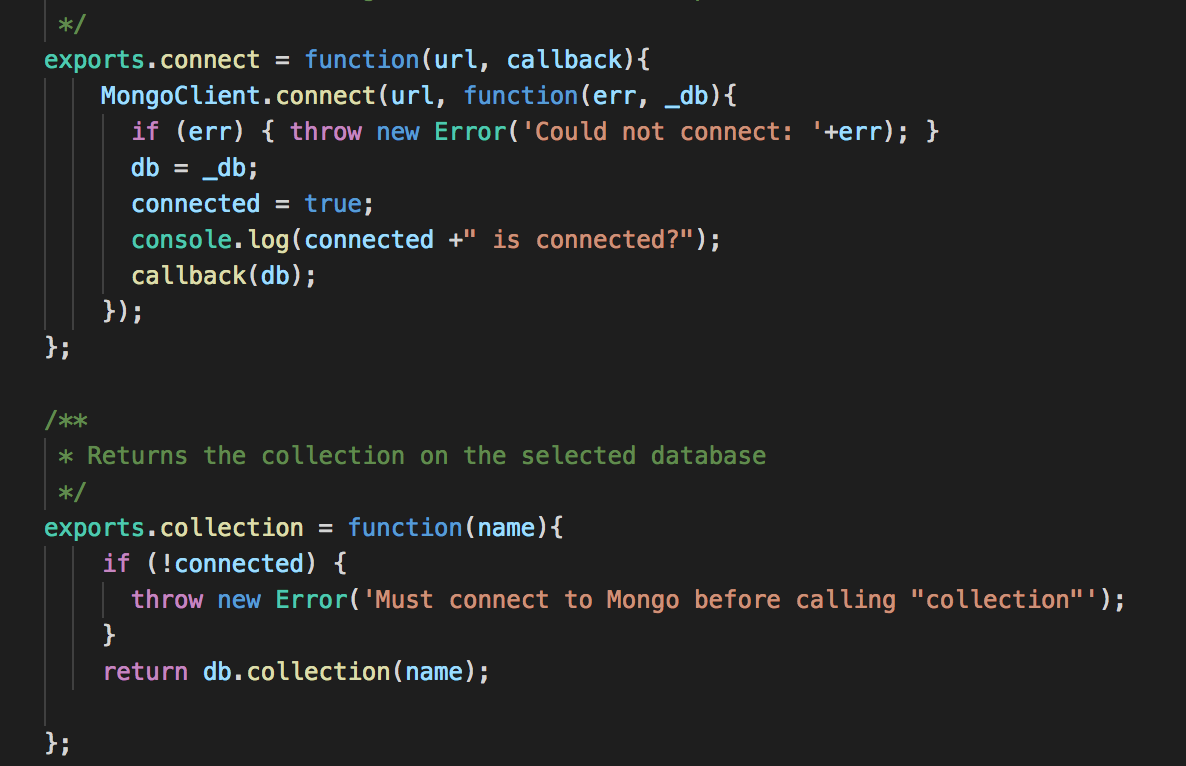
In the kafka-backend directory, we will create a folder called “services” and we will create a file to every functionality where the function handlers are handled. Here, we will create a connection where the request and responses are pushed and pulled into the topics where the client and server can communicate in this way.

The topics which we create for our functionalities are given below:

* Signup\_topic
* Login\_topic
* Logout\_topic
* Uploadfiles\_topic
* Getfiles\_topic
* Createdir\_topic
* Stardir\_topic
* Sharedir\_topic
* About\_topic
* Creategroup\_topic
* Addmember\_topic
* Showmembers\_topic
* Deletemember\_topic
* Deletegroup\_topic
* Response\_topic

The messages for all the functionalities are pushed into these topics. And the responses from the back-end services are pushed into the response\_topic where the client can pull these messages from it.

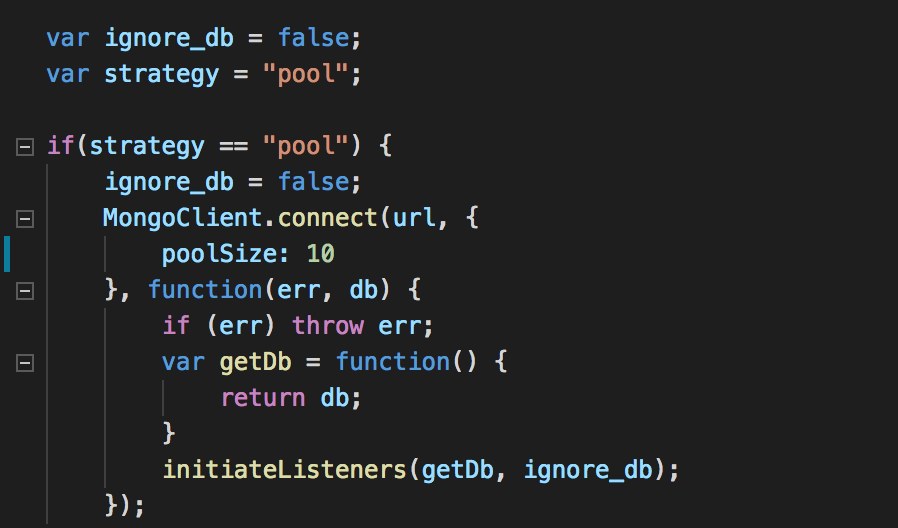
We need to connect our kafka back-end to the MongoDB and it is done this way:



**Connection Pooling:**

We are implementing our own connection pooling in this lab. We will be using round robin scheduling for the allocation of connection. Here, we will create an array with limited number of connections. We create 3 connections in the array and if a new connection is requested. It will be allocated to the connection in a round robin manner. Similarly, the new connections are allotted to the other places in the array.

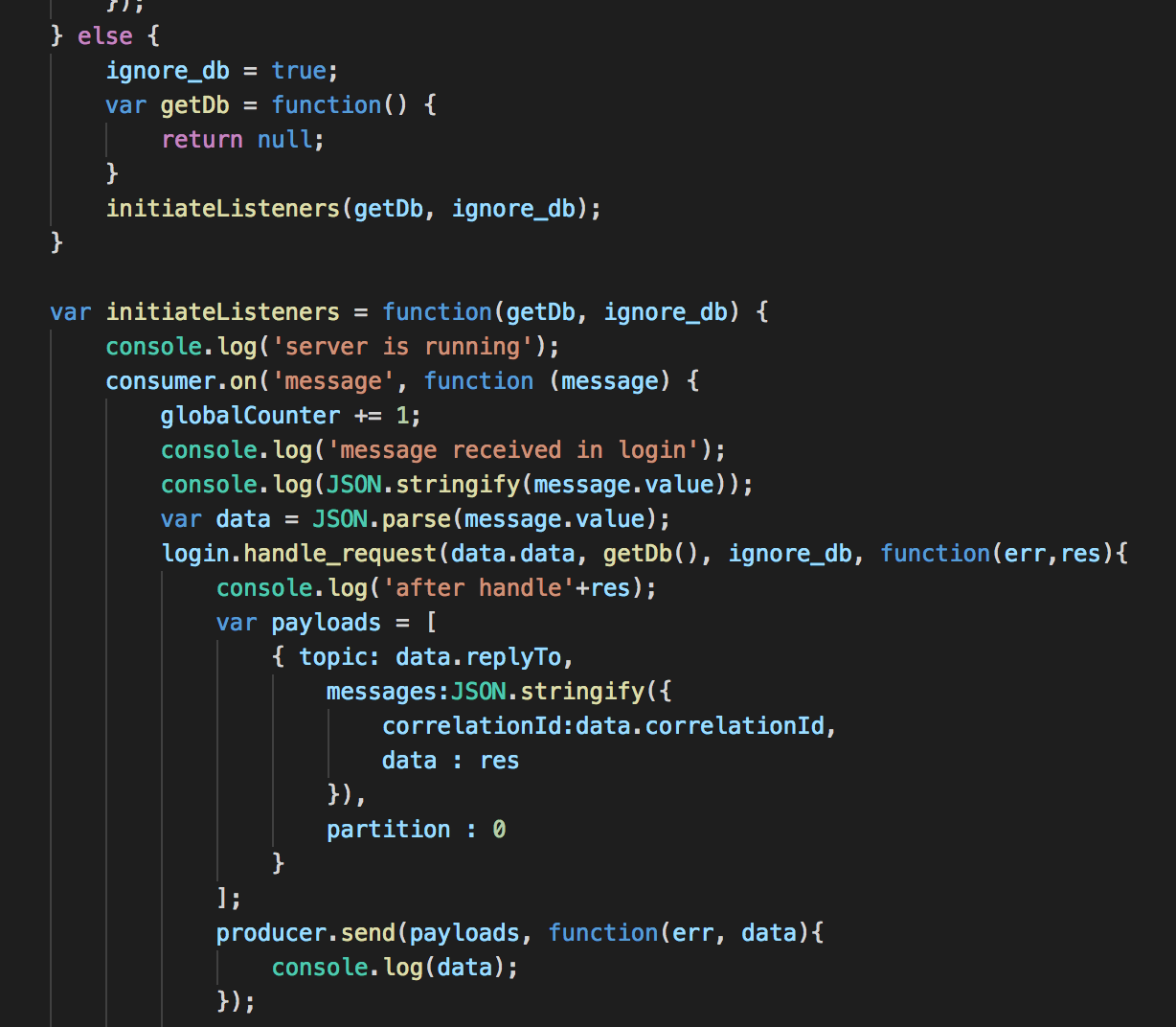
We will be creating if else cases of our connection pooling for the performance analysis. The first case would be the default in-built MongoDB connection pool. When the strategy is default pool the connection pool should be as following:



When the strategy is our own pool, then the connection pool should be as following:

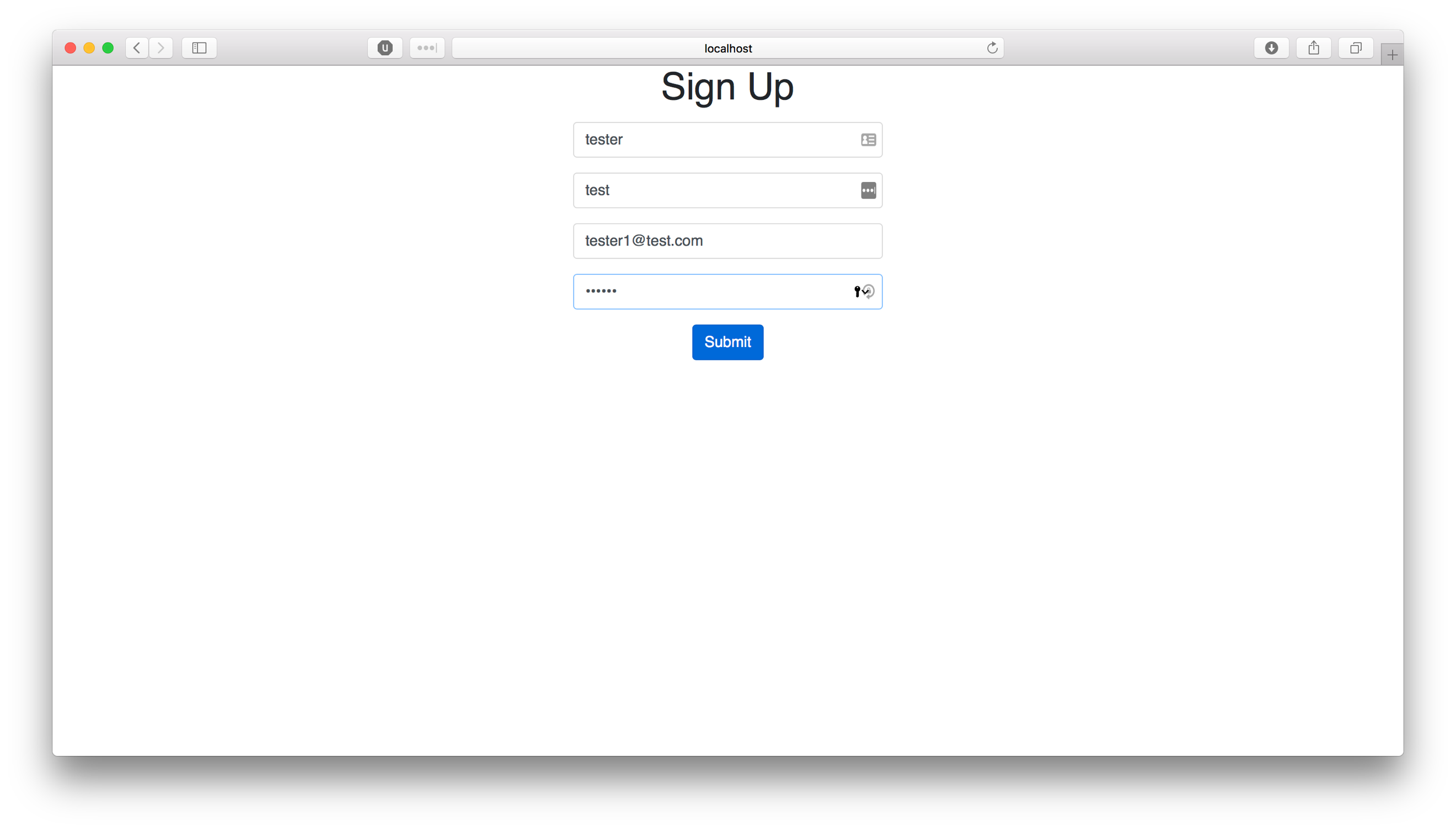


If there is no pool implemented, that means a new connection is established for every request. This is not a good way to process the requests. Then the pool connection will be as:

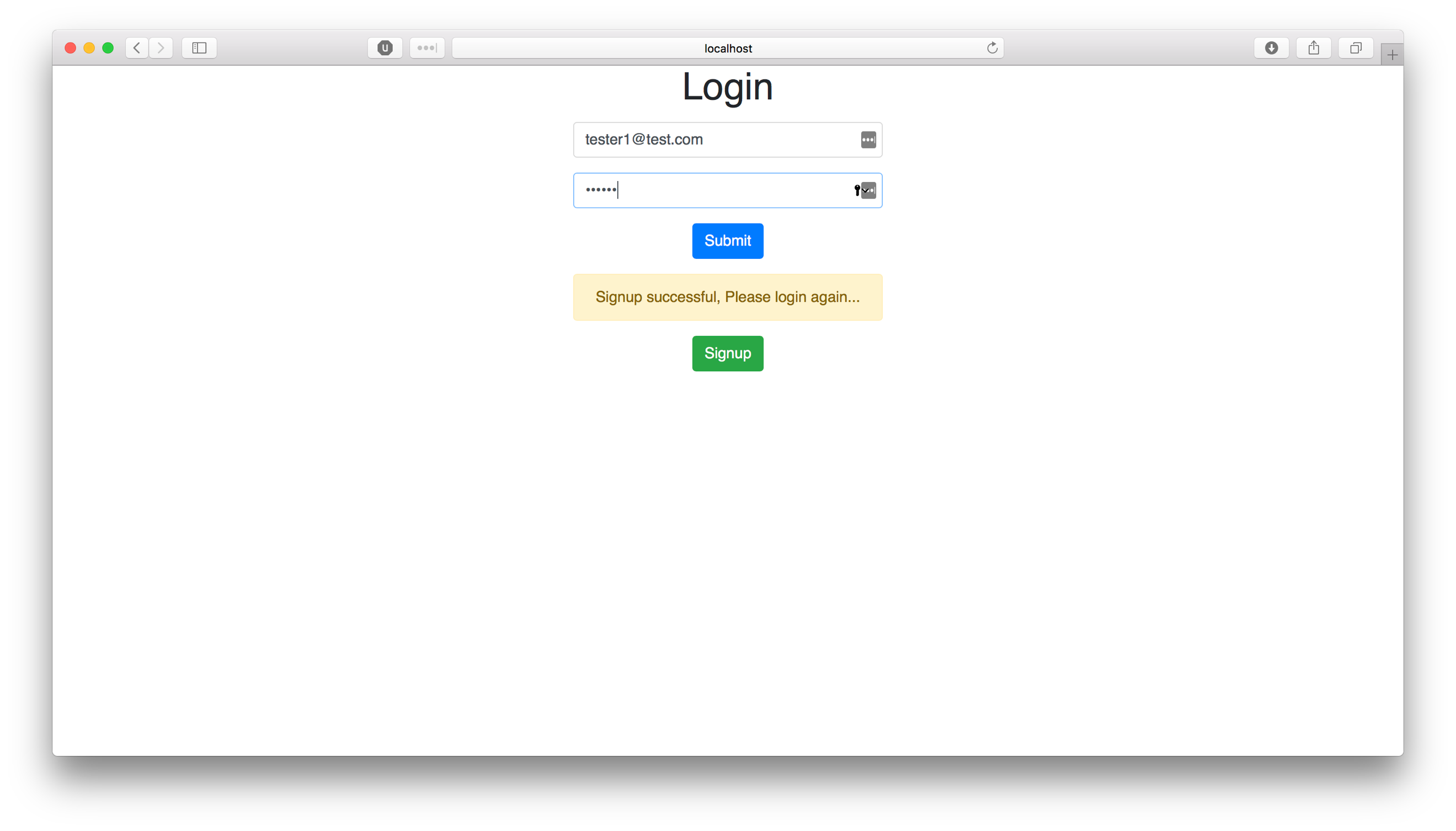


**Client:**

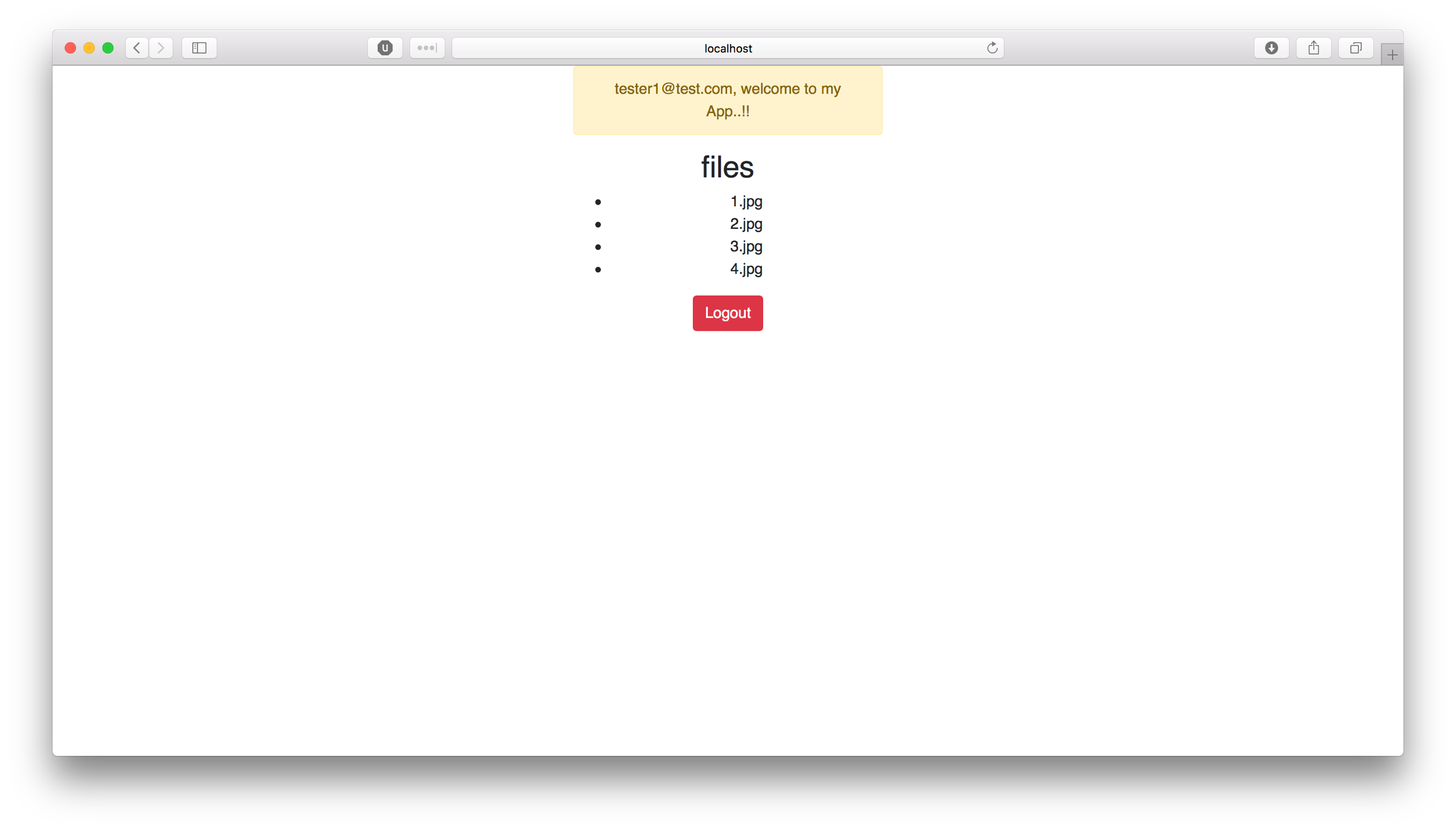
The below image is a screen capture of the signup page on client side services:



The below image is the screen capture of the login page:



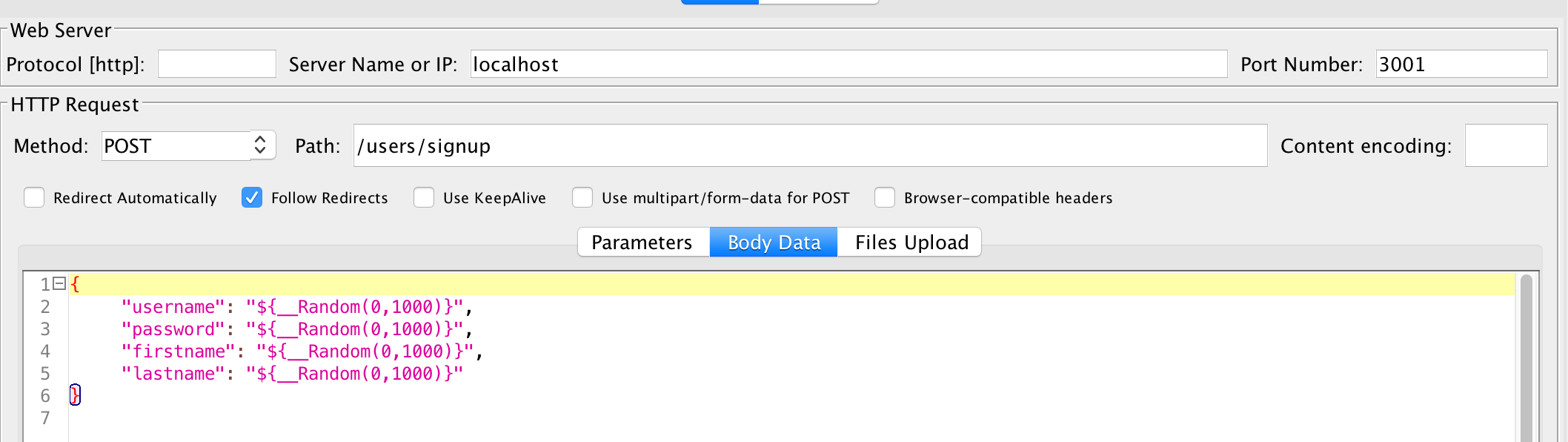
The homepage of the client side services where we can list all the files is:



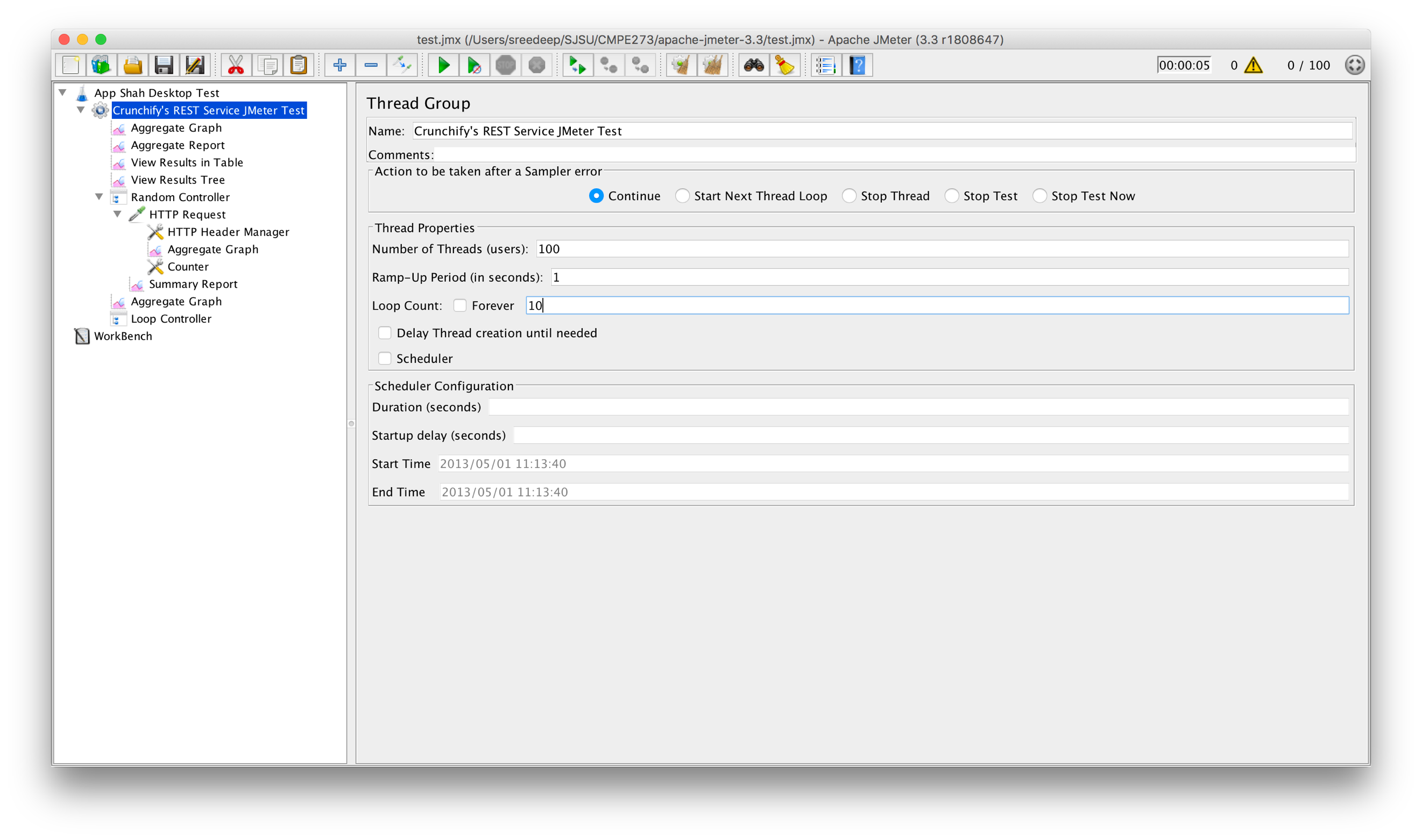
**Performance:**

**Testing with JMeter:**

We will test the server using JMeter, and we will pass the random parameters for our requests in the HTTP body data in this way:



We can set the number of concurrent users according to our requirements.



Test the server for **100, 200, 300, 400 and 500 concurrent users**

1. without connection pooling

Without connection pooling, the average time for **100, 200, 300, 400 and 500 concurrent users** is as follows:

|  |  |
| --- | --- |
| Number of concurrent users | Average Time(ms) |
| 100 | 349 |
| 200 | 593 |
| 300 | 1383 |
| 400 | 3728 |
| 500 | 5349 |

**(b)** With your own implementation of connection pooling

With our own implementation of connection pooling, the average time for **100, 200, 300, 400 and 500 concurrent users** is:

|  |  |
| --- | --- |
| Number of concurrent users | Average Time(ms) |
| 100 | 274 |
| 200 | 455 |
| 300 | 433 |
| 400 | 588 |
| 500 | 740 |

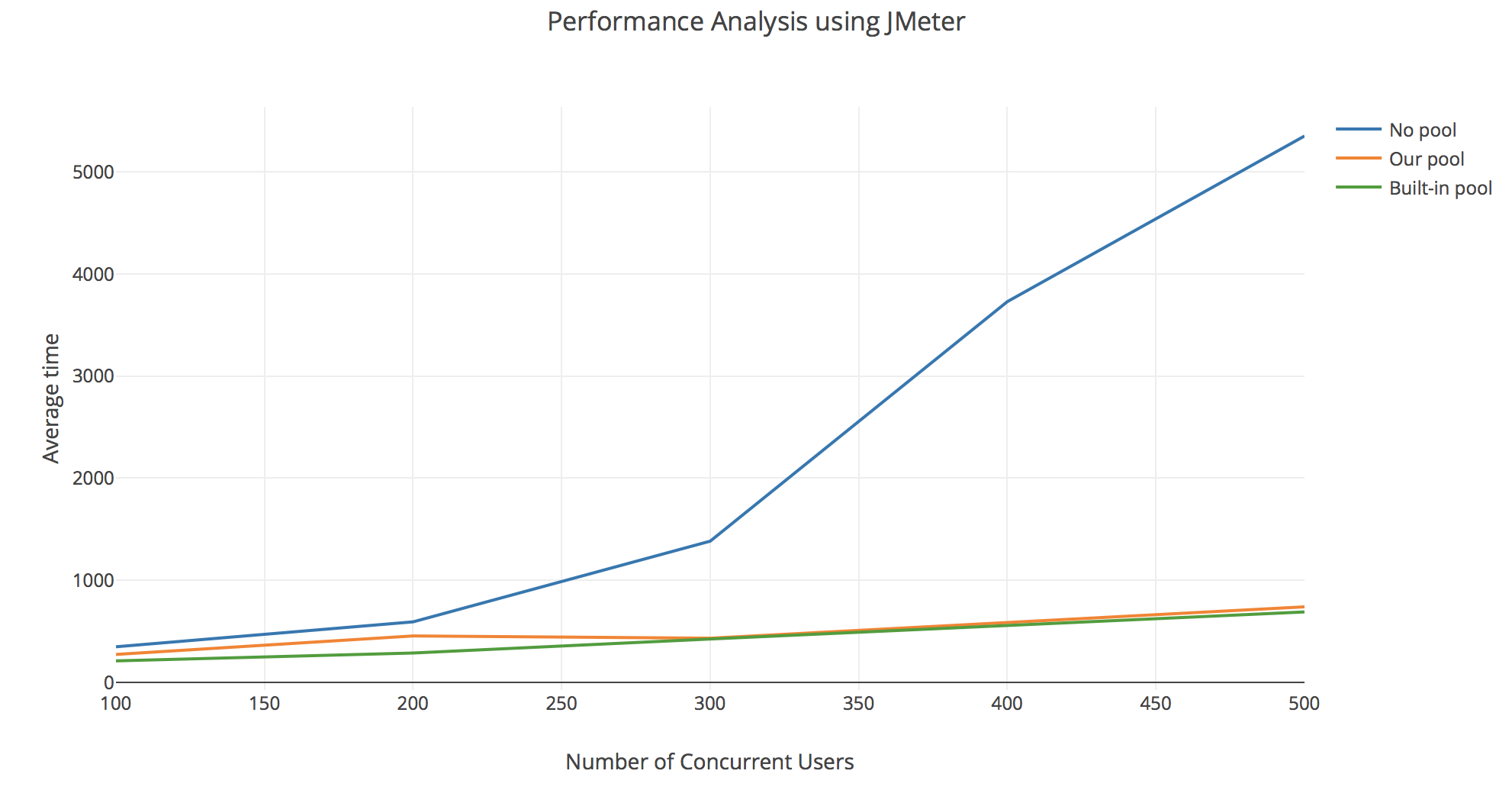
1. DB provided connection pooling

With DB provided connection pooling, the average time for **100, 200, 300, 400 and 500 concurrent users** is as follows:

|  |  |
| --- | --- |
| Number of concurrent users | Average Time(ms) |
| 100 | 211 |
| 200 | 288 |
| 300 | 425 |
| 400 | 555 |
| 500 | 690 |

**Draw the graph with the average time, your analysis of the graph on why, why not and how in the report**

A graph is drawn with the average time for these three cases, it is formed as shown below:



From this graph, it can be stated that DB provided connection pooling is ideal for the performance of the server to be at its best. DB provided connection pooling is much better than our own connection pooling because our pool is maintained in a round robin manner, where the 4th connection has to wait at the 1st connection until it is completed. Unlike our connection pool, the DB provided connection pool is better at optimizing the requests for the server.

**Questions:**

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

In Lab1 the authentication and session management is handled via passport and mysql. In Lab2 the authentication and session management is done using passport and mongodb. The key difference in the approach used in the second lab is that the sessions are persisted to mongodb by passport whereas the sessions were not stored in lab1. This gives us an added advantage where we can technically restore the session even if the user clears their browser cache and can keep a record of all the sessions initiated by the given user.

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

With the introduction of kafka, the performance numbers are increased exponentially. The key factor for this added performance is because of the design choices made as part of the kafka. Instead of sending each message individually, kafka prepares a microbatch (which can contain 1000s of messages) and sends this microbatch across the network. Since majority of time is spent in sending the message across the wire, kafka tends to increase the performance by sending a much larger number of messages in a single batch (there by increasing the throughput)

**3. 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**

MySQL is much more suitable for structured data whereas MongoDB is more suitable for unstructured (hierarchical) data. If given an option to implement both MySQL and MongoDB, I would choose to store structured data such the username, password etc for users, list of files for a user, list of groups, members etc in Mysql and I would choose to store additional information about users, files, groups etc in MongoDB.