**CMPE273 - LAB 1**

**CALCULATOR**

1. Introduction.

The goal here is to create a calculator application that does 4 basic operations, addition, subtraction, division and multiplication.

1. System design.

This working calculator is made up of two parts, Client – Front-end (React), and Server – Express (Node).

Here we do all the data input value in a nice front-end, the constraint is we should have two operands and one operator, we can’t cascade many functionalities, this is its design.

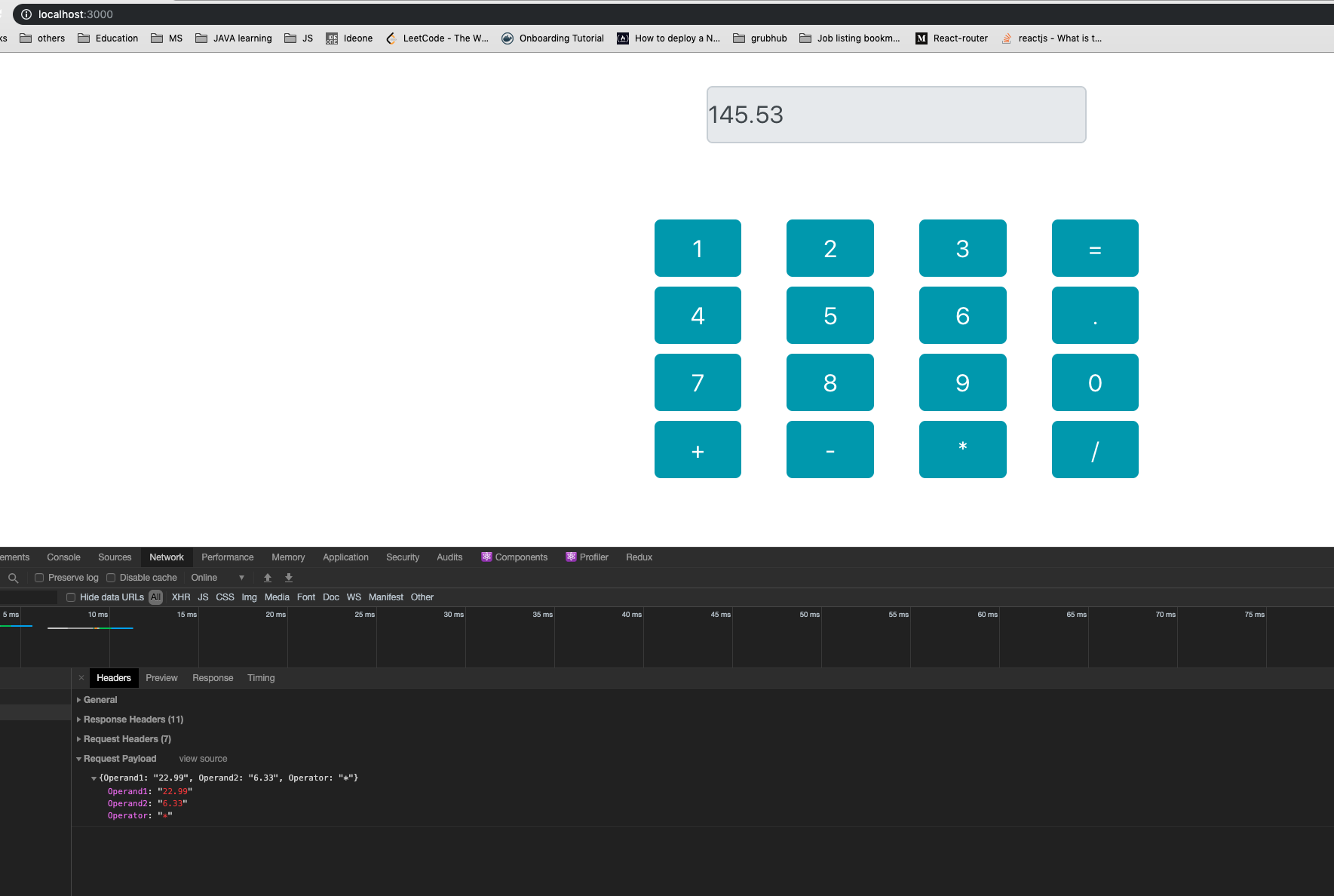
We process the entered input as a nice JSON, that can be readily used by back-end to service and return a proper response.

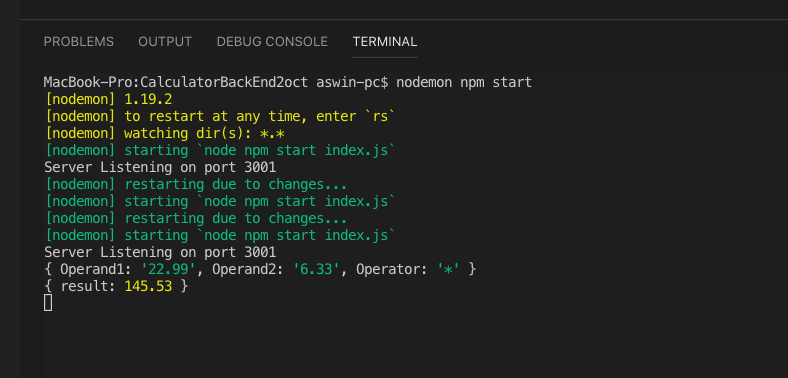
{ Operand1: '22.99', Operand2: '6.33', Operator: '\*' }

1. Results.

UI & Backend.



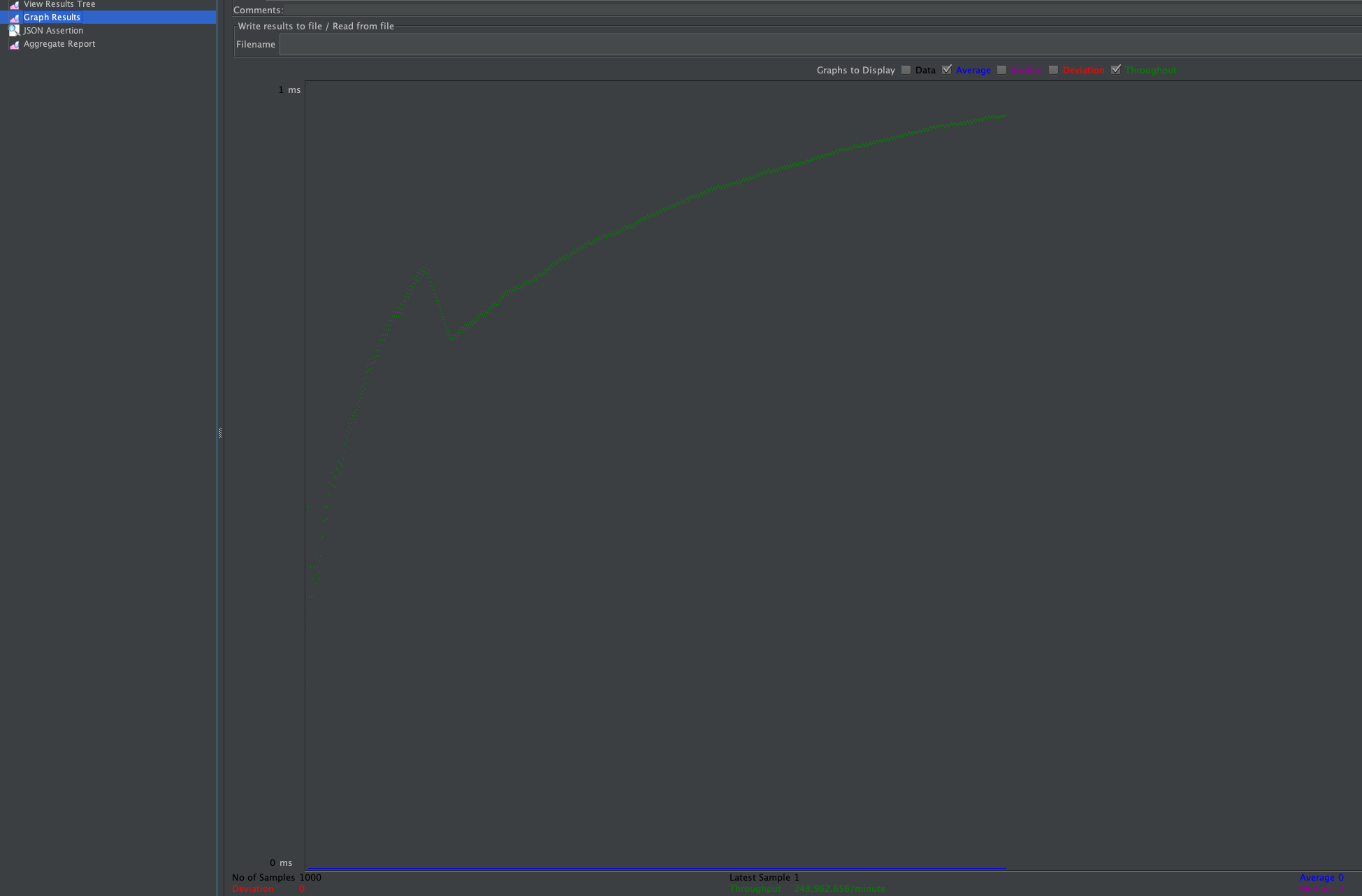


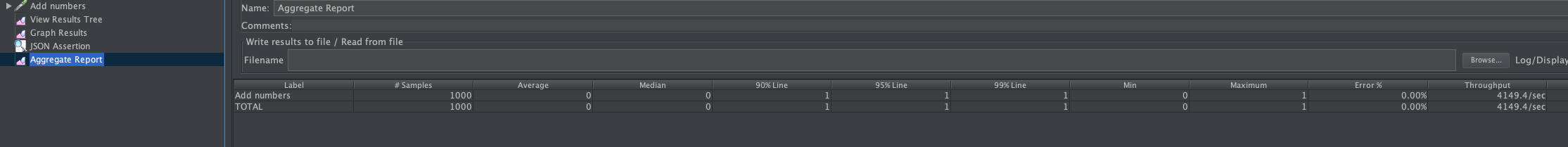


J-Meter testing.

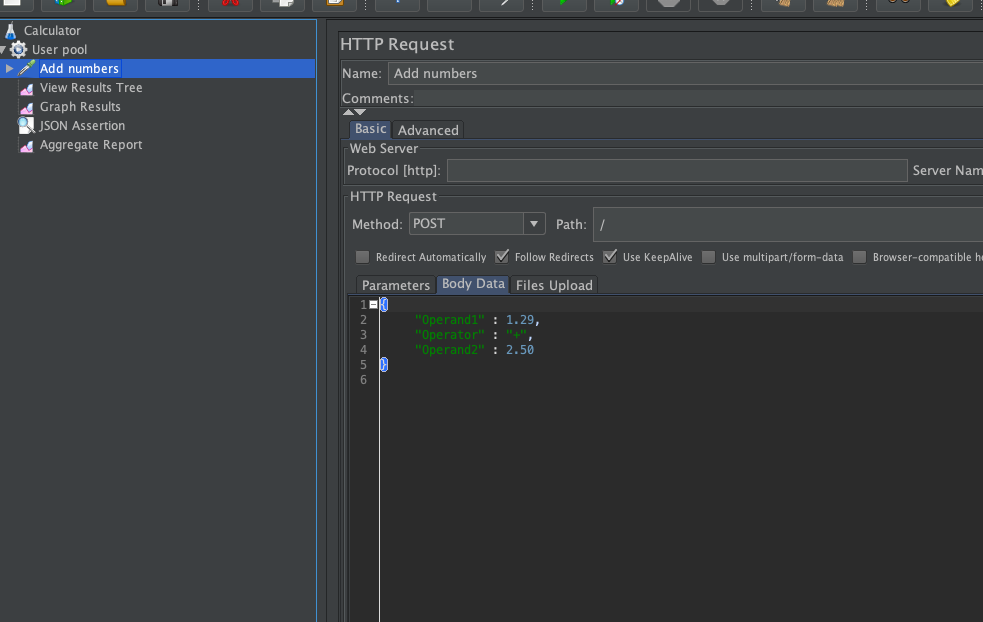
1000 calls, one thread (Addition).

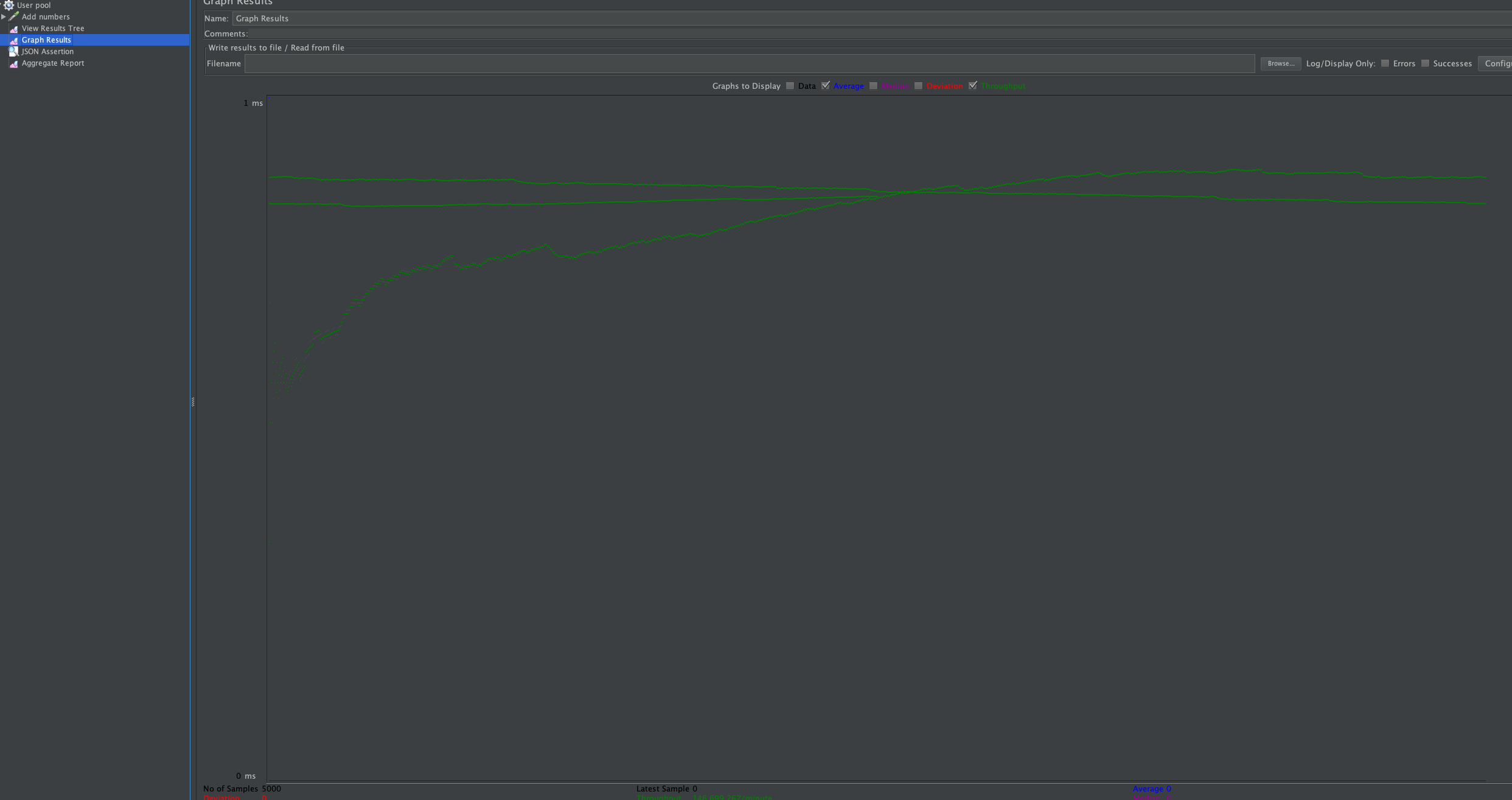


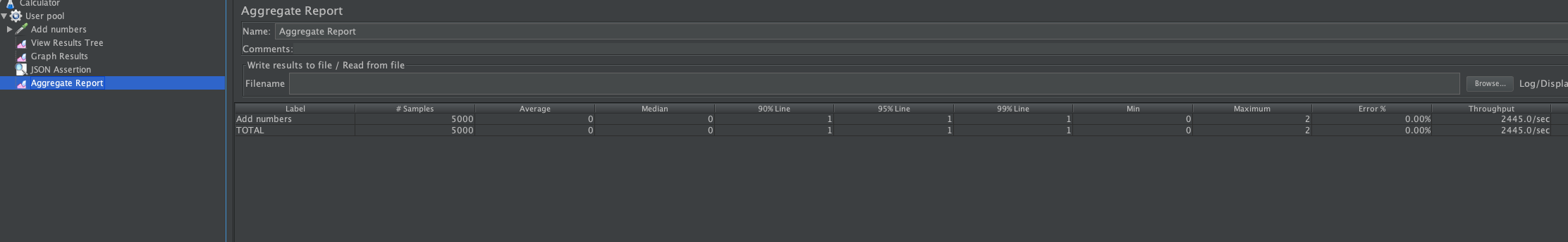




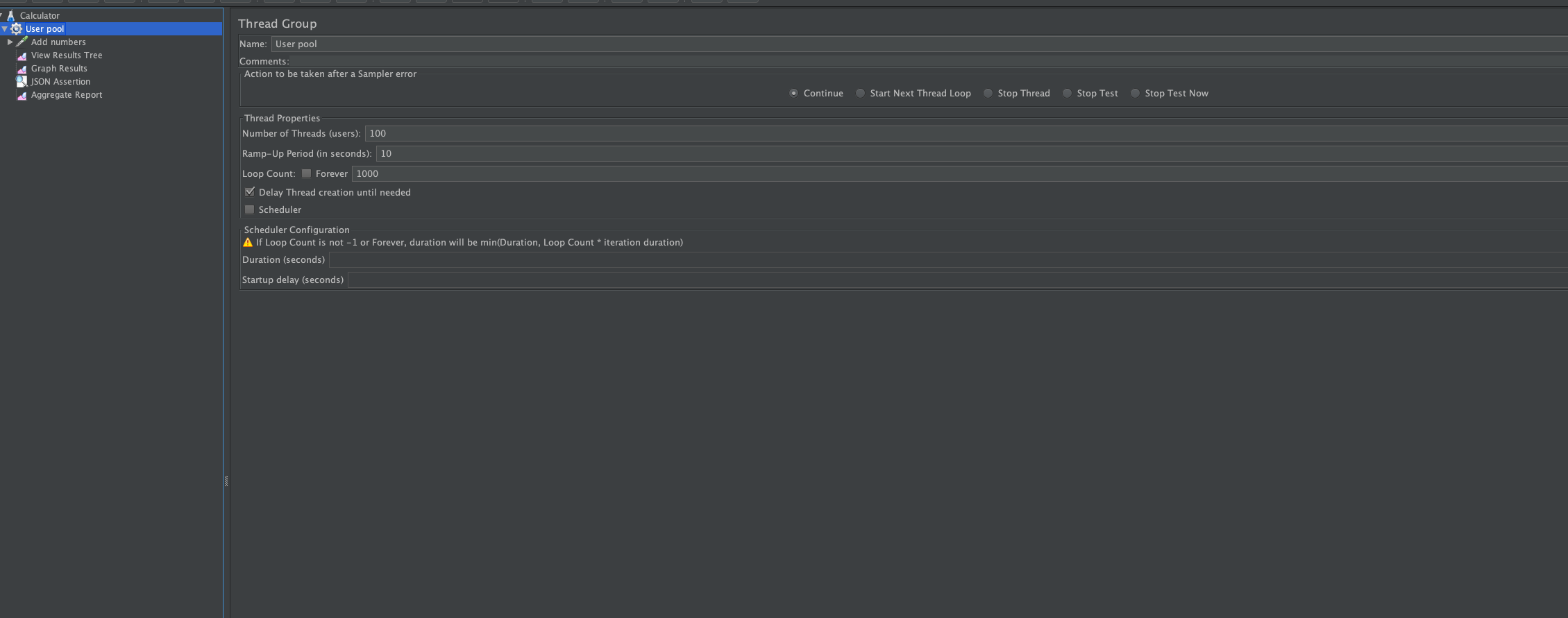
5000, one thread (Addition).

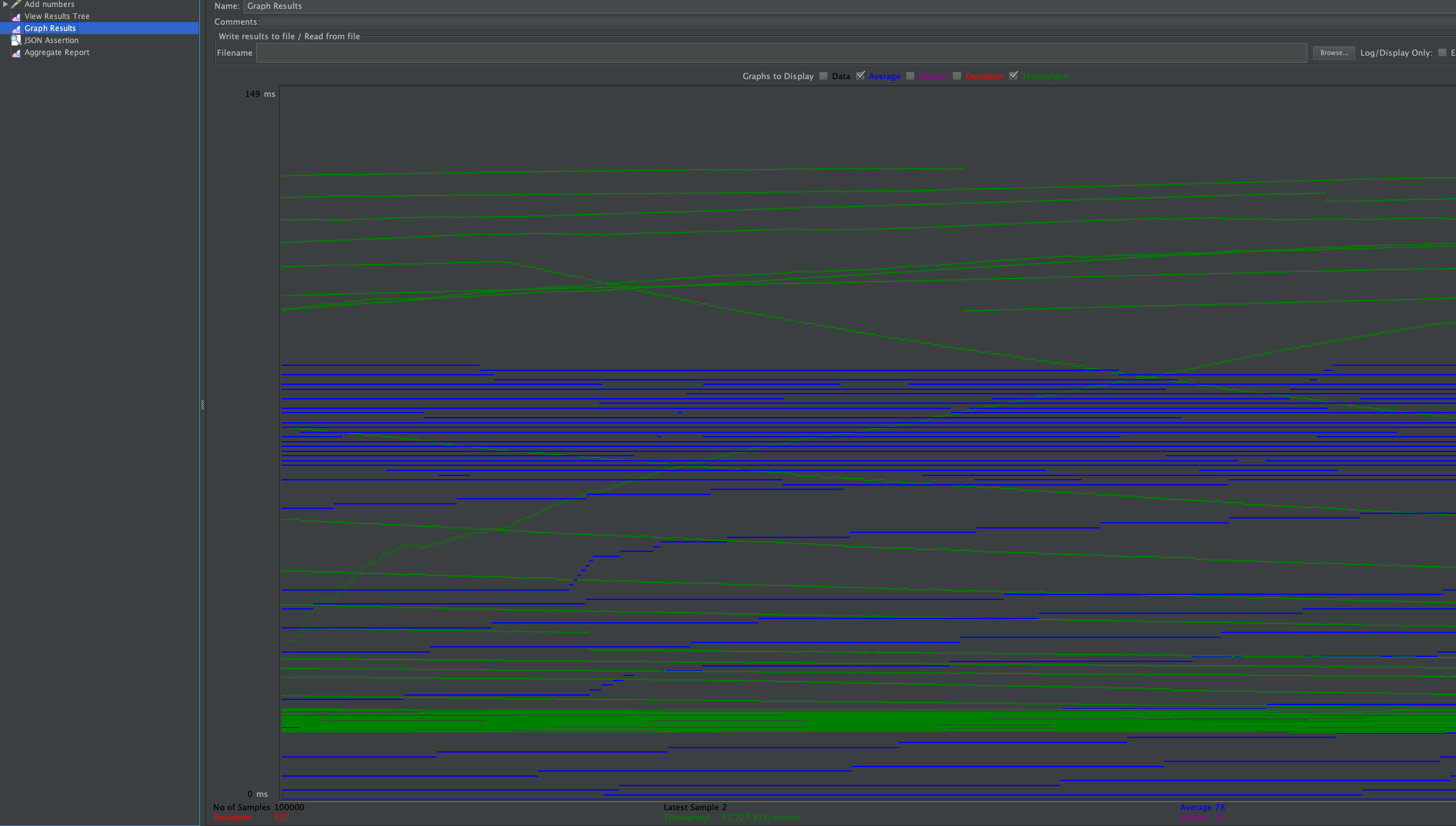


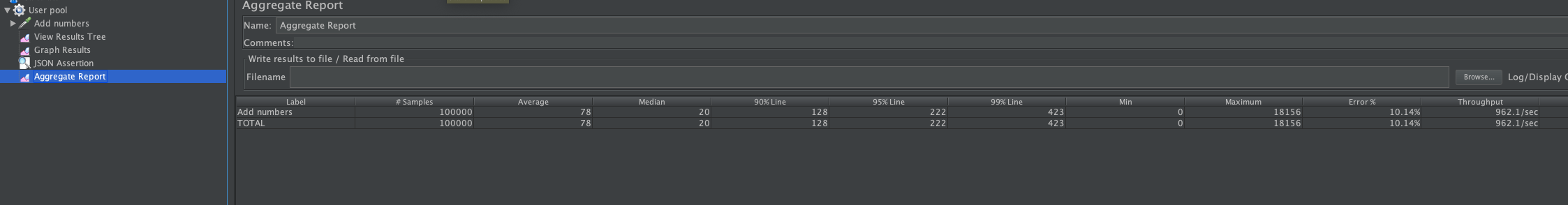




1000 calls, 100 concurrent users (Addition)







Here in this case, localhost is not a very powerful machine, it can’t handle 10000 connections, when all request server (same port) at once. So, there is failure percentage of 10% at peak traffic.

`java.net.SocketException: Resource temporarily unavailable (connect failed)`.

1. Performance.

There are various ways in which we can do the application for a calculator. The way I have designed is that to do all the validations and reductions in the front-end itself and just sending the crucial data (Operands and operator) to back if they are valid. In this way the performance of the application will be pretty good.

1. Results.

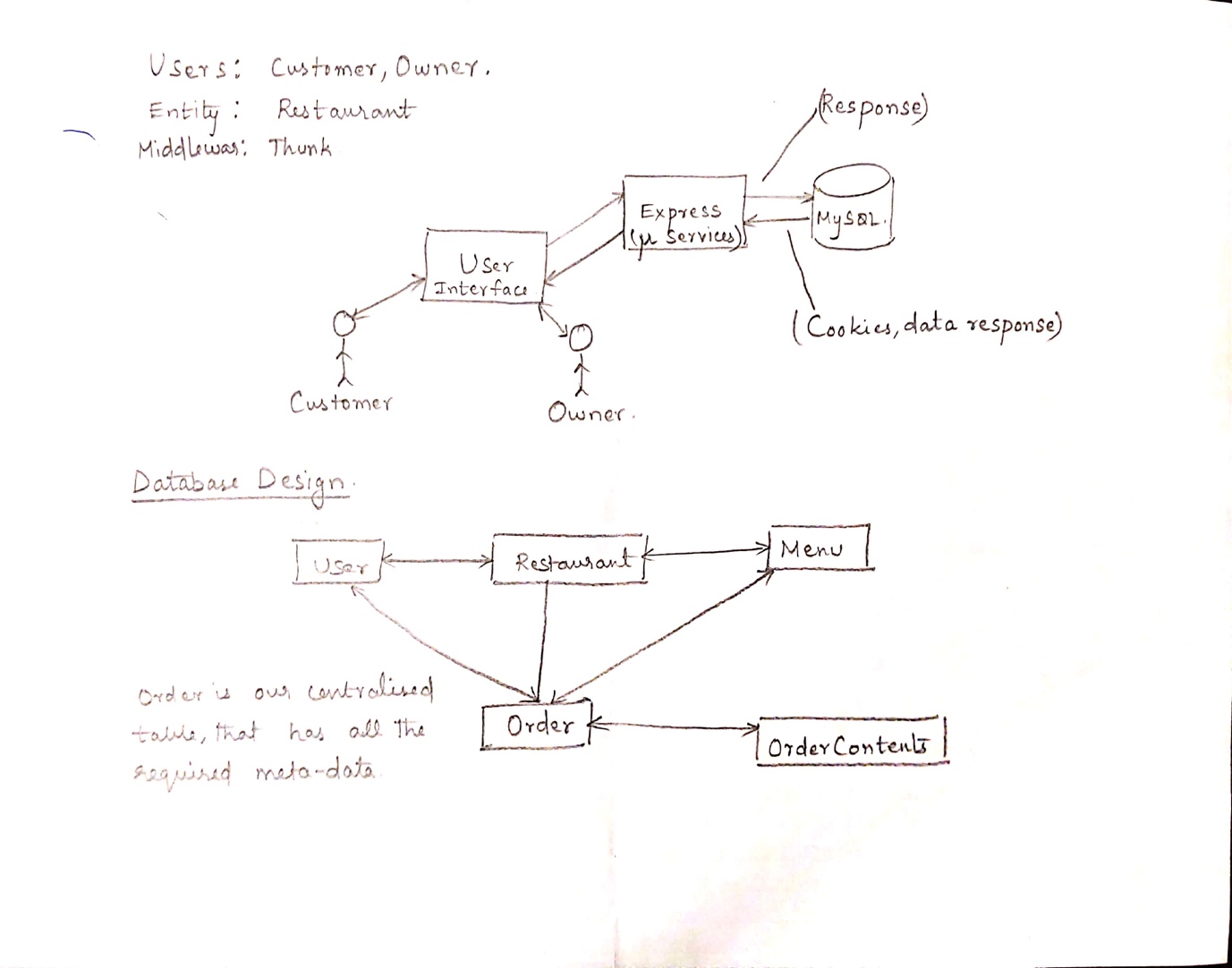
Thus, we have calculator front-end and scalable backend to service that front-end. We handled all the validations in front-end and allowing the access to server only if the request is valid. Regarding the performance, it is running without any problem when we do 1000 and 5000 calls, but when there are 10 users and each do 1000 calls, our local machine port can’t handle that load and give exceptions when servicing those specific requests.

**Part-B Grubhub**

1. Introduction.

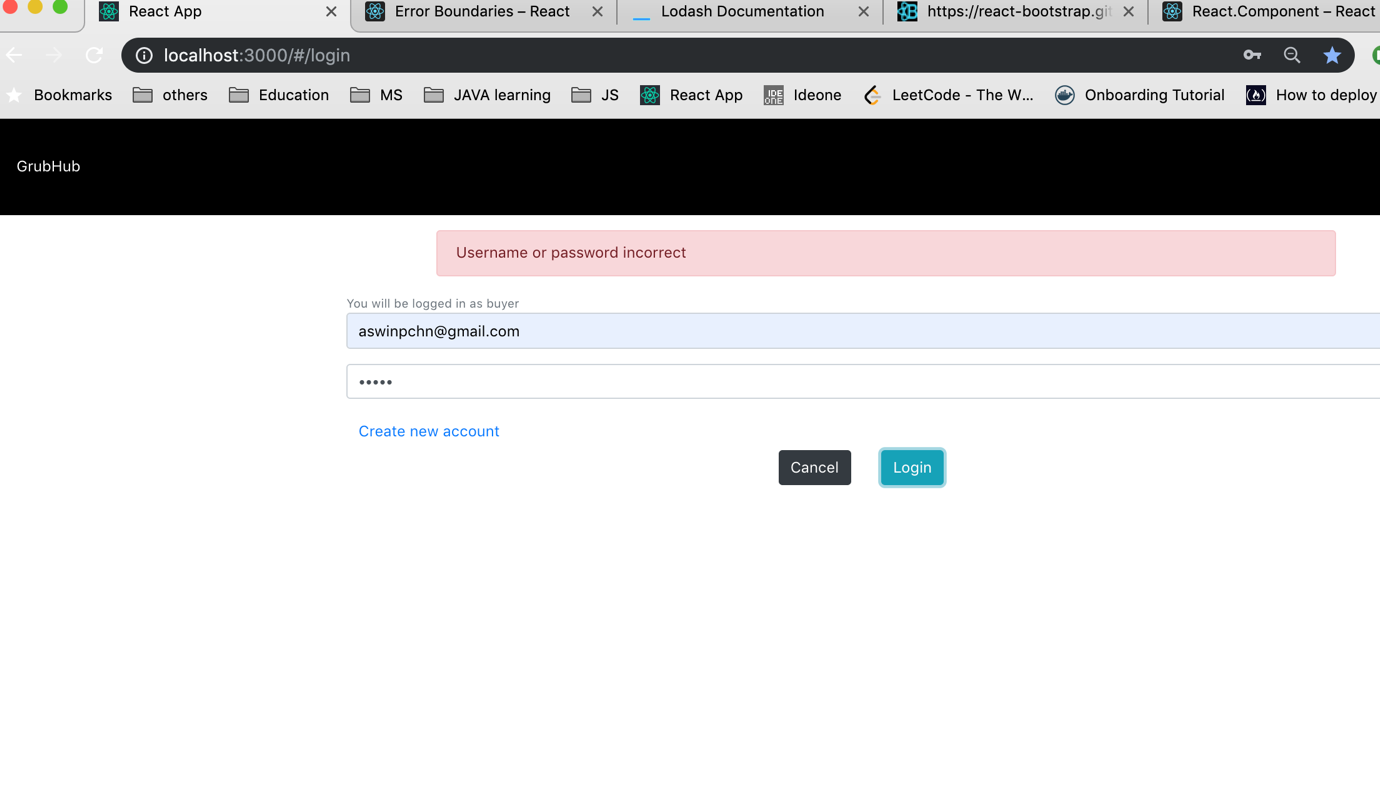
The goal is to create a complete food ordering application starting from customer side order to restaurant side order management. To satisfy this purpose, we are using React, Redux, MySQL and Express (Node).

1. System design

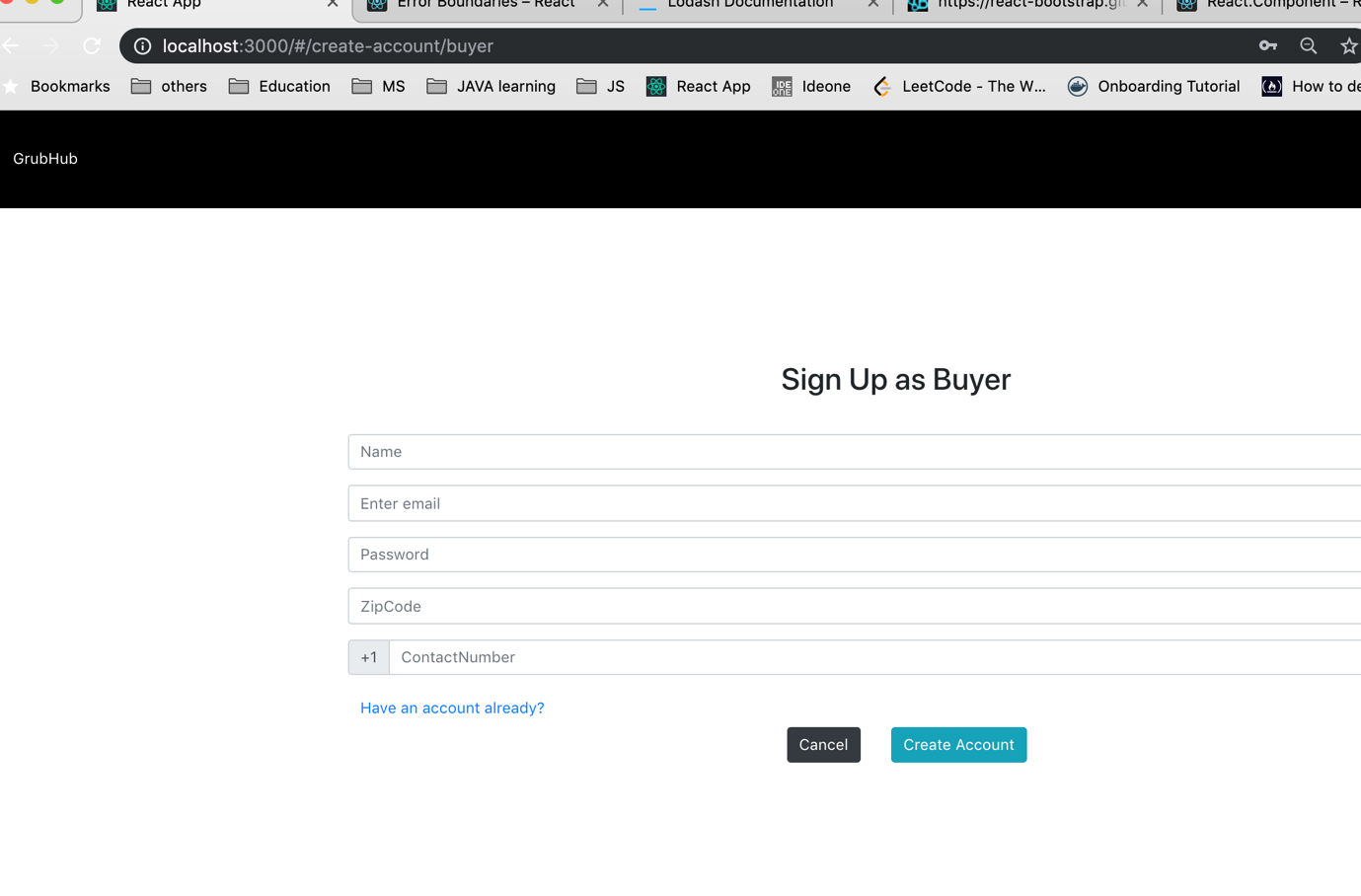


1. Results

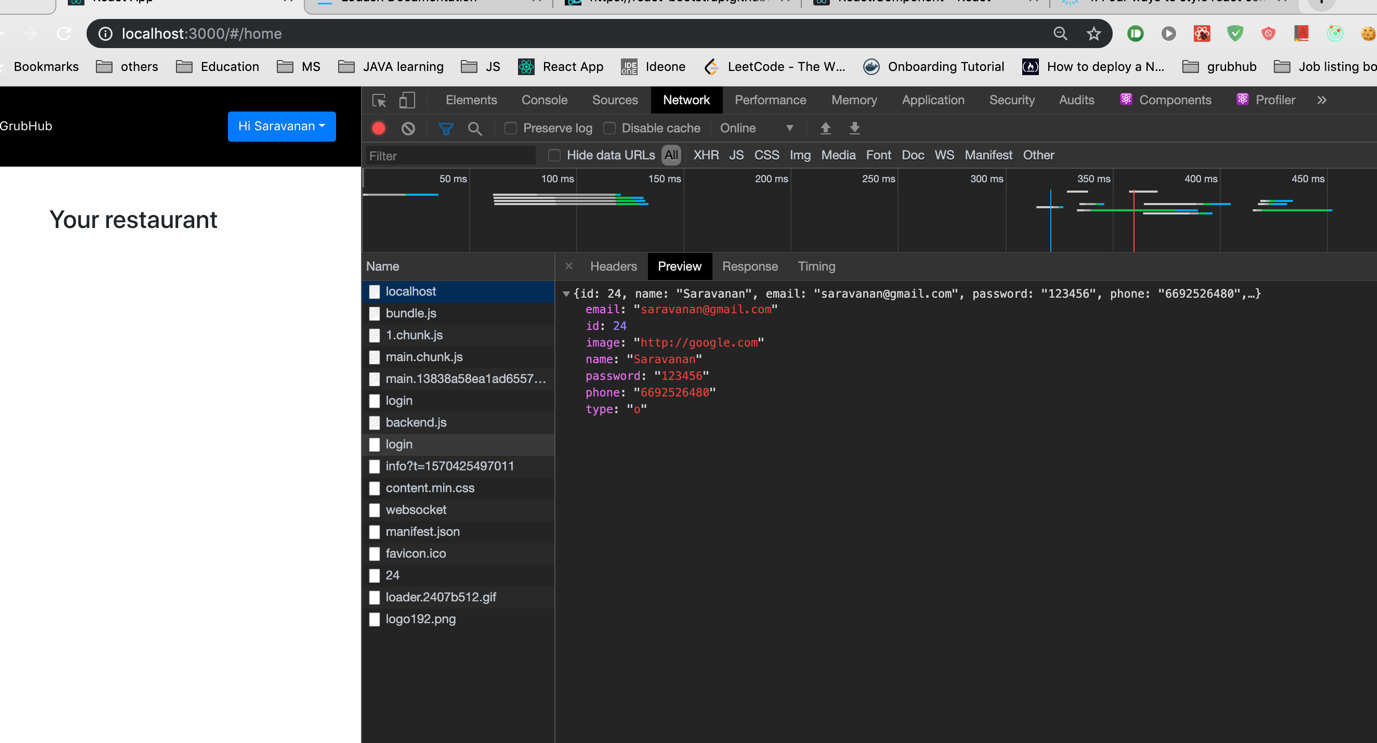
Login page (Wrong password)



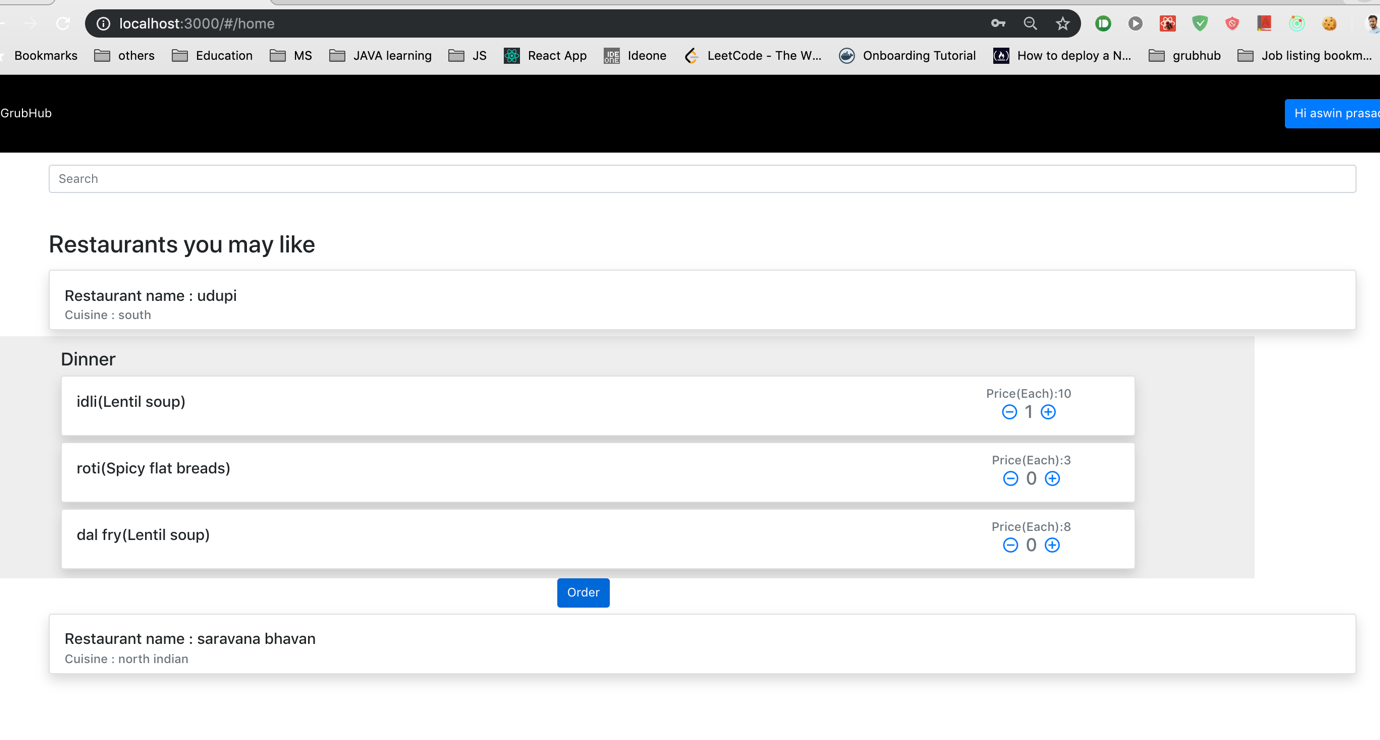
Signup page



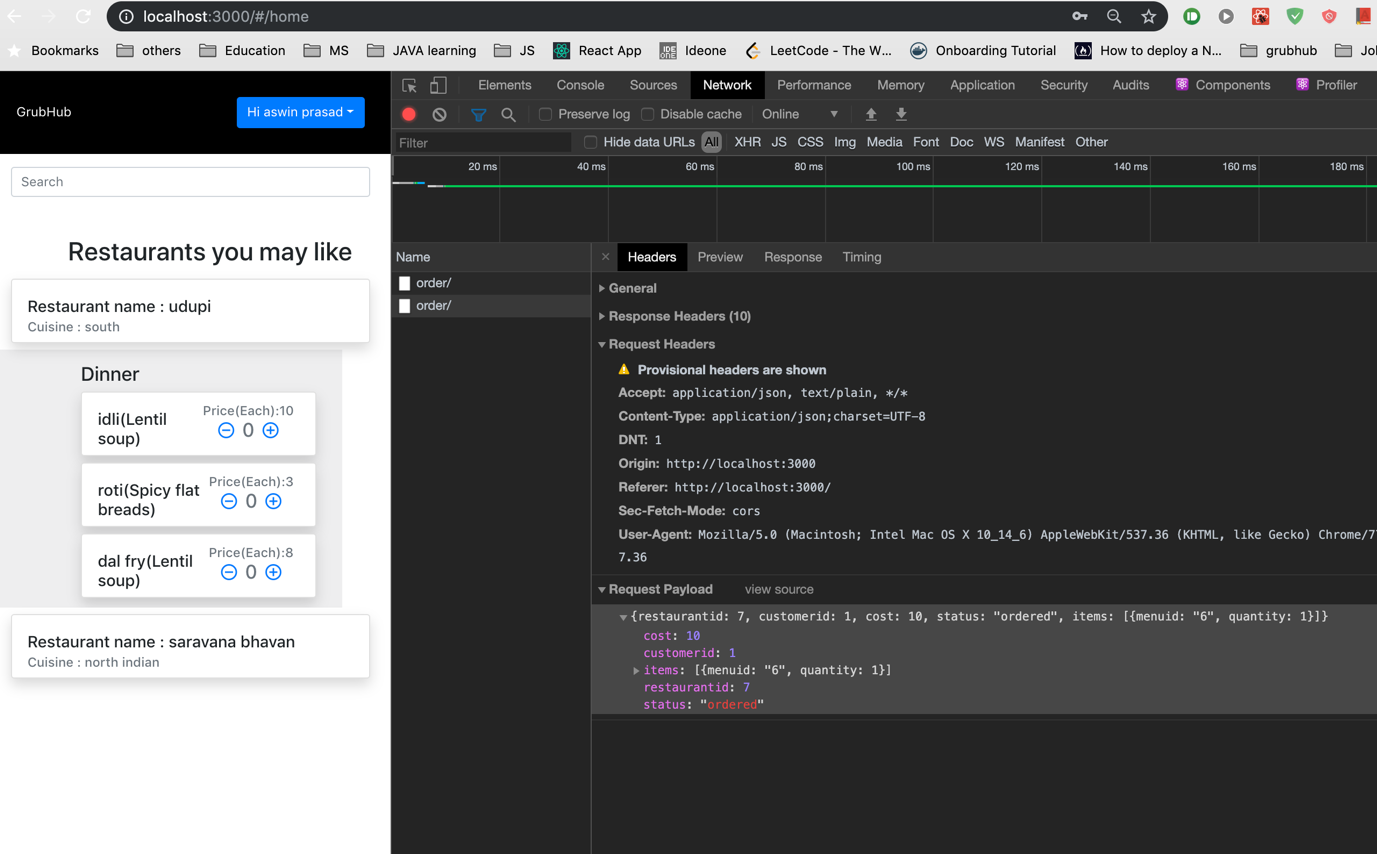
Successful login

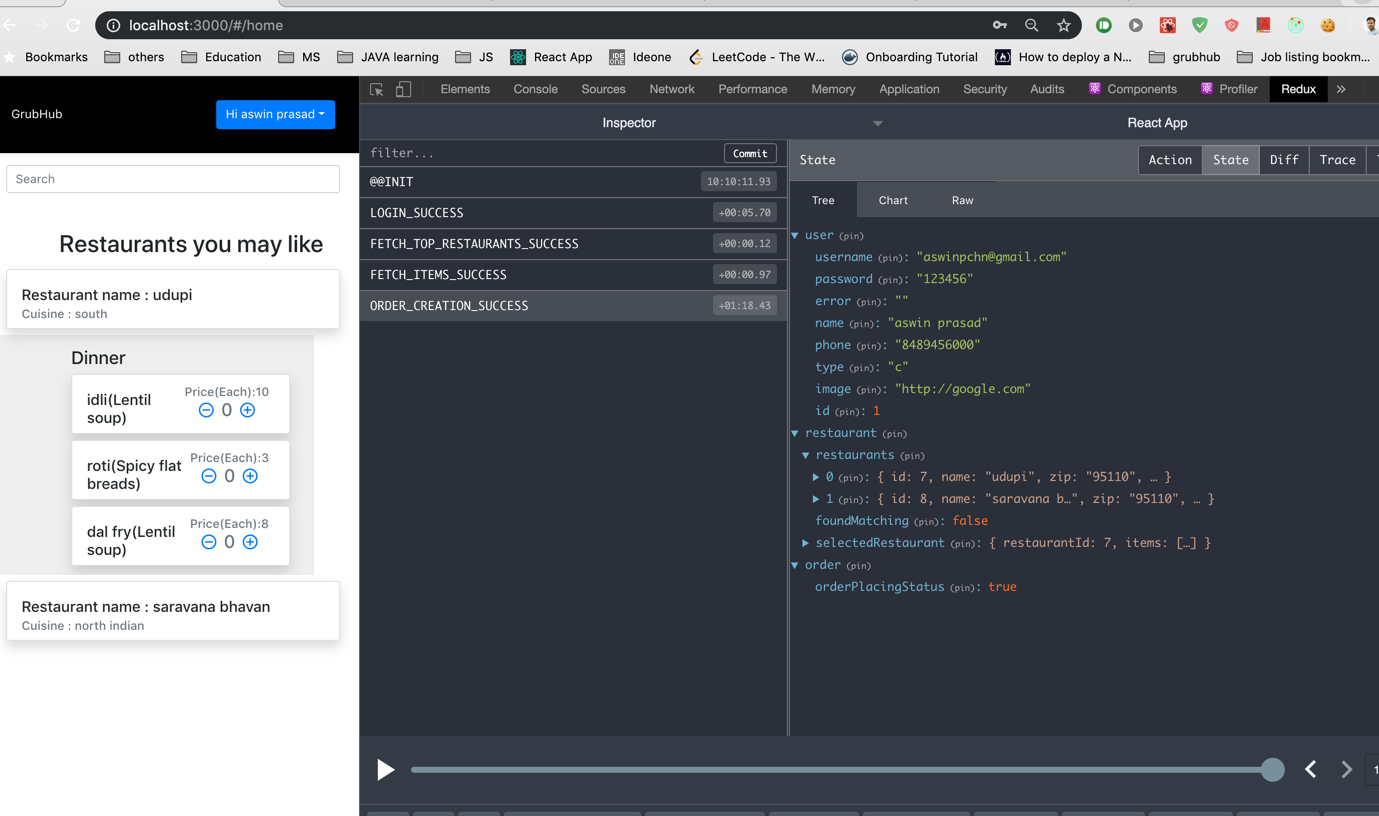


User ordering page

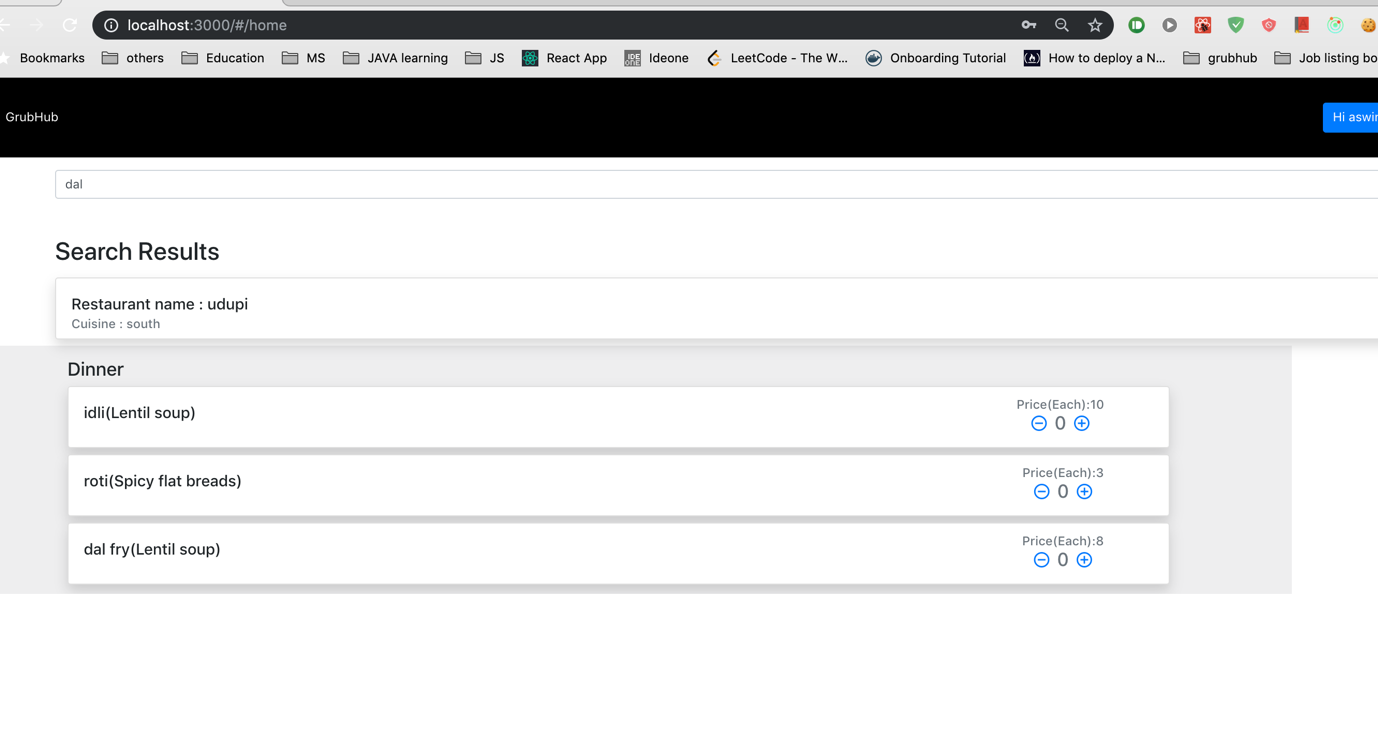


Successful order page

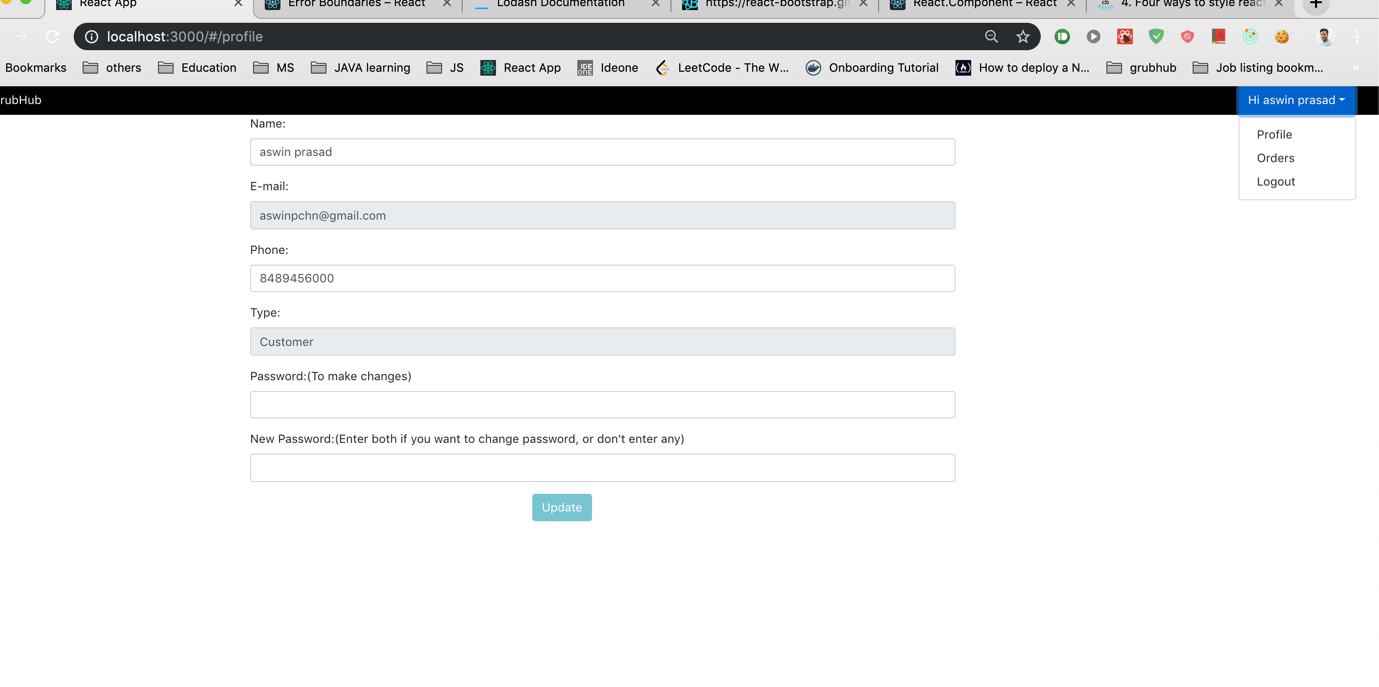




Food search page – By name



User details update page.



Backend API documentation. (Postman documentation)

<https://documenter.getpostman.com/view/945162/SVtSUpBr?version=latest>

1. J-Meter testing (Without pooling).

Code that was used for testing.

router.get('/owner/:id', (req, res) => { // Get a restaurant by ownerid.

    if(!req.params.id)

    {

        res.writeHead(400);

        res.end("wrong parameters");

    } else {

        let user = req.params.id;

        let responsePromise = dbCall(`select \* from restaurant where ownerid=${user}`);

        responsePromise.then((response) => {

            if(response.length !== 1) {

                throw "no restaurant";

            }

        res.writeHead(200, {  // //res.type('json')  // This also will work similar to setting content type application/json

            'Content-type' : 'application/json'

        });

        res.end(JSON.stringify(response[0]));    // We can't send JSON directly we have to change it to string using stringify

        }).catch((error) => {

            if(error == "no restaurant") {

                res.writeHead(404);

                res.end("restaurant not found");

            } else {

                res.writeHead(500);

                res.end("db error");

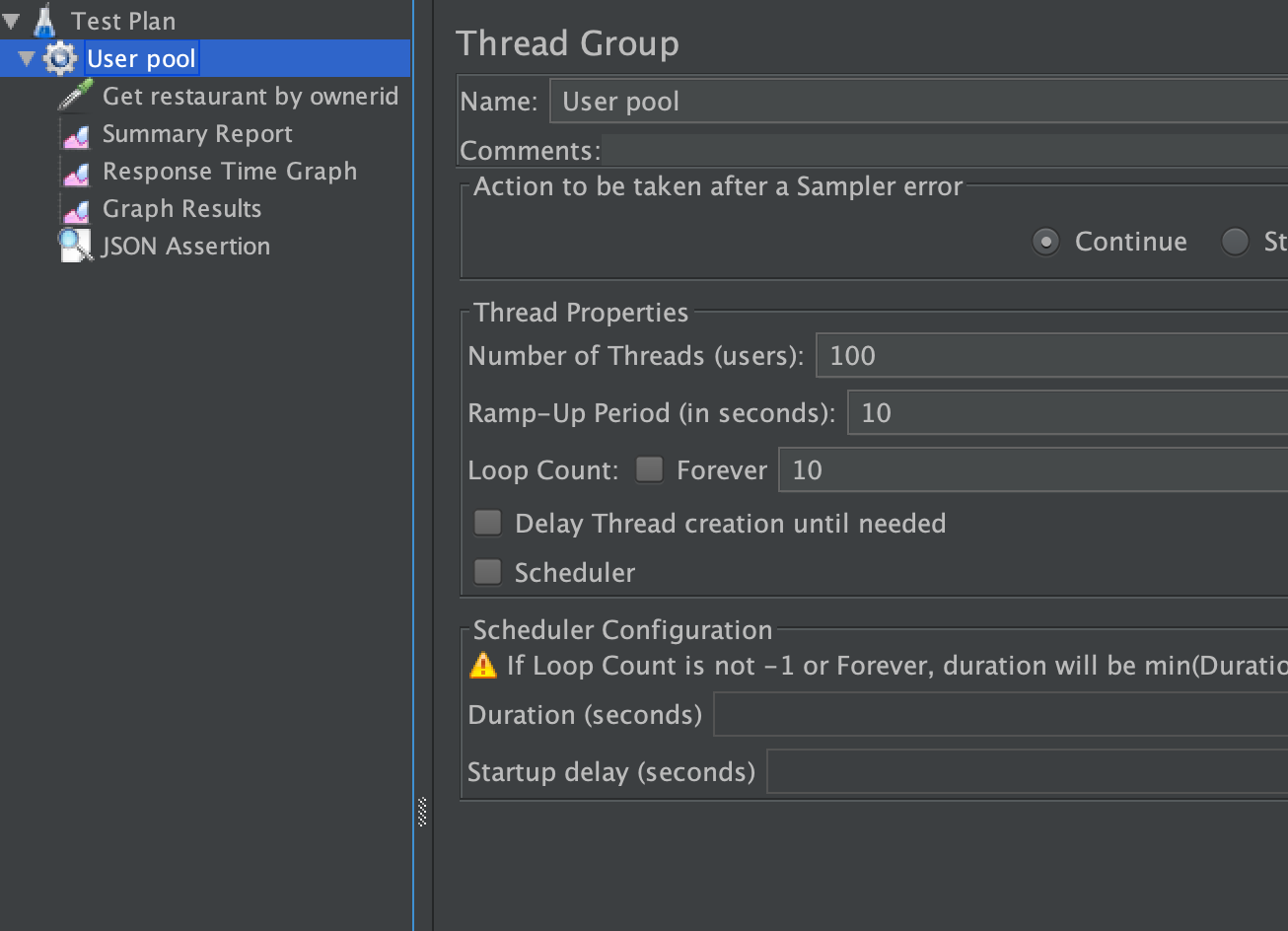
            }

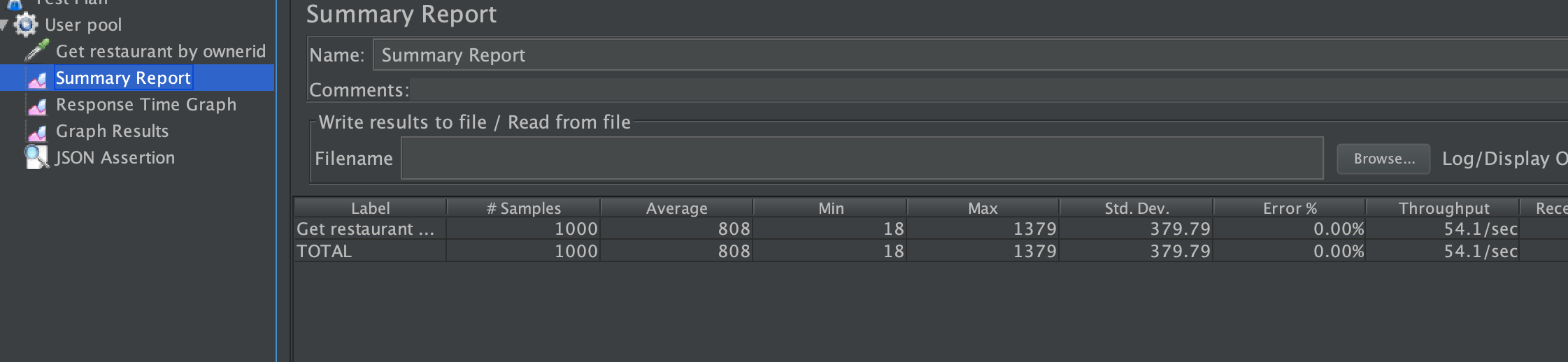
        });

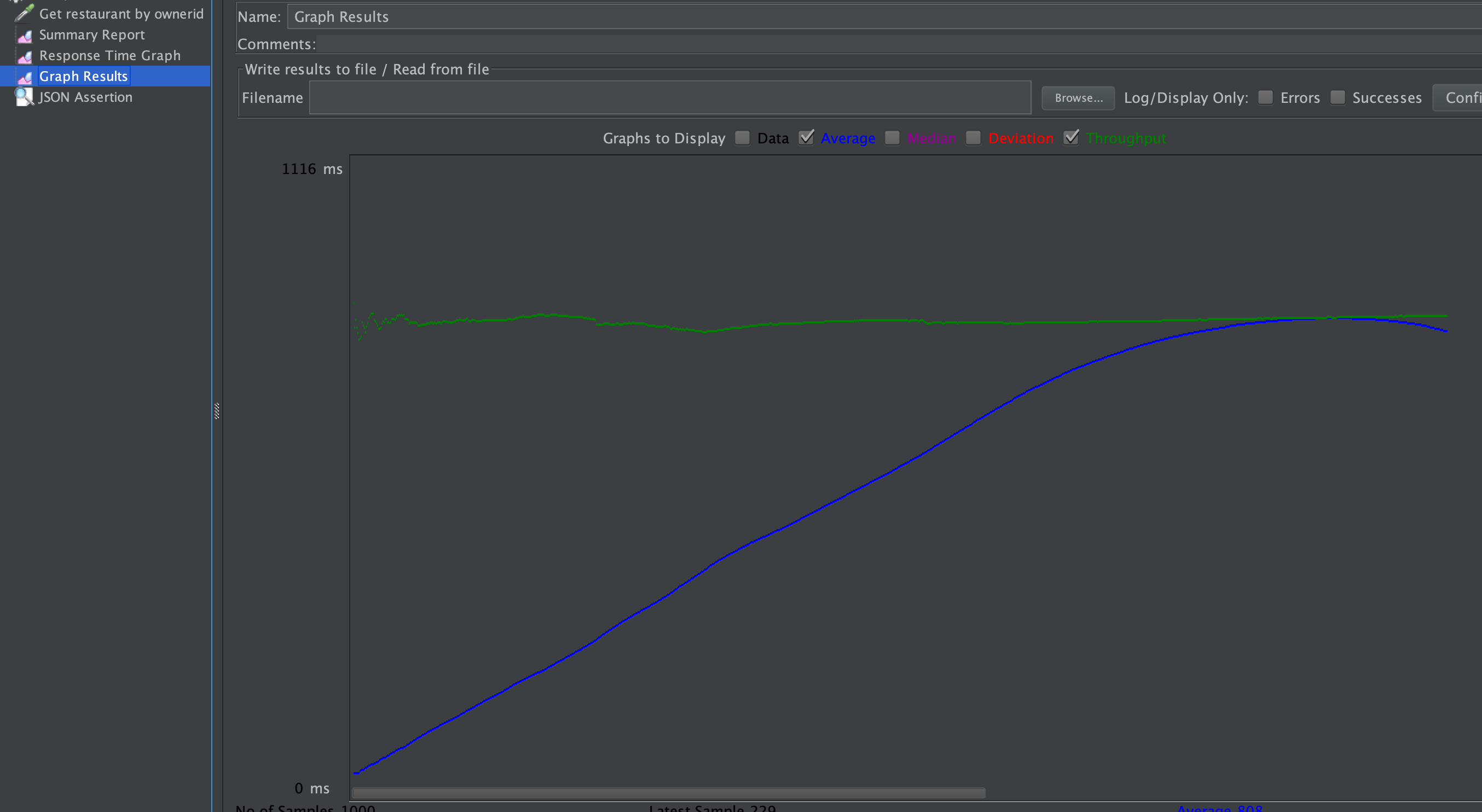
    }

});

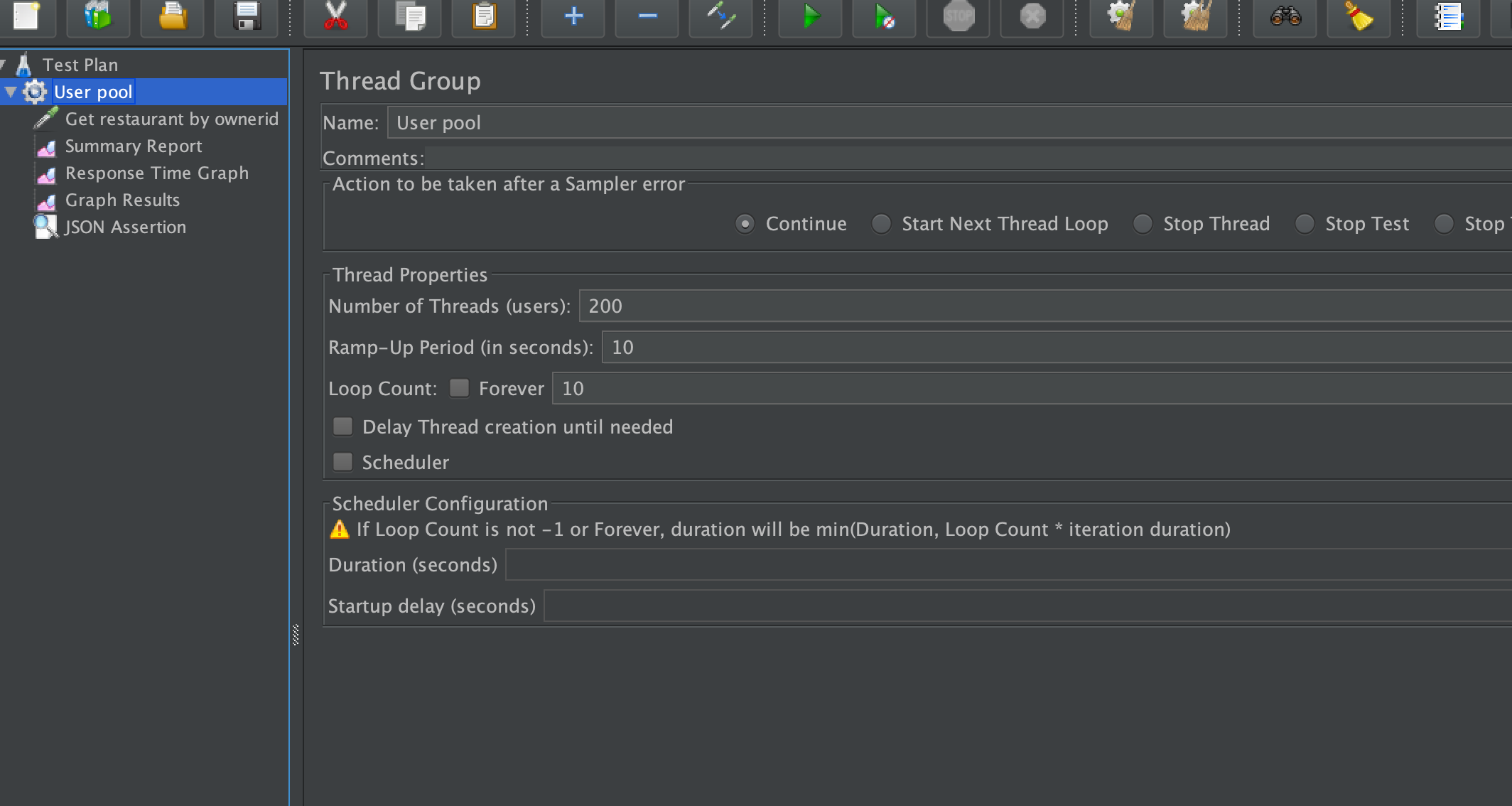
1. 10 Calls, 100 concurrent users.

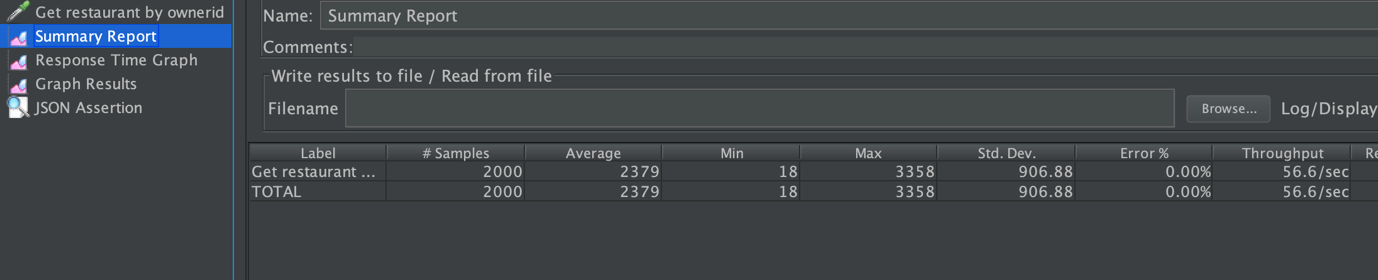


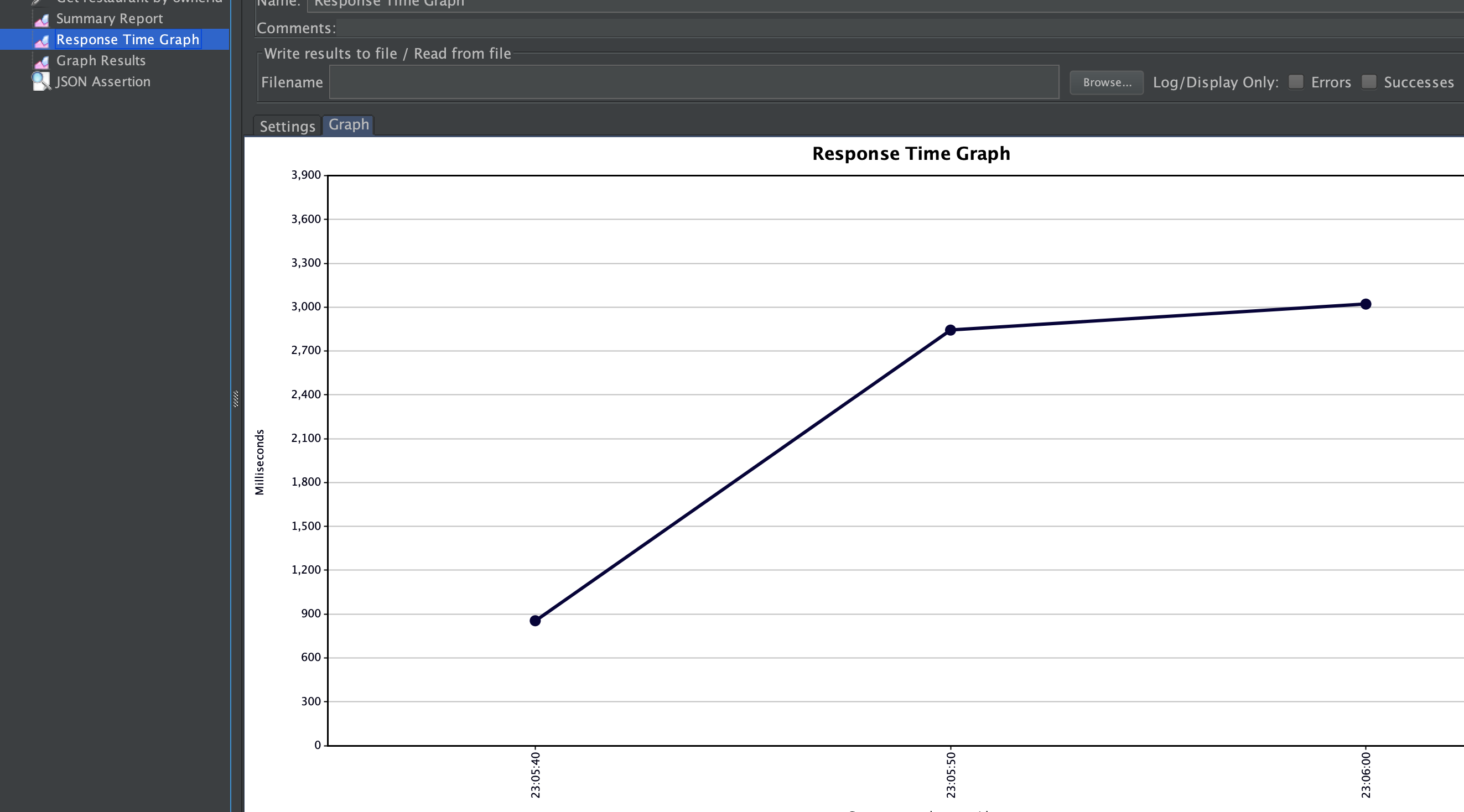




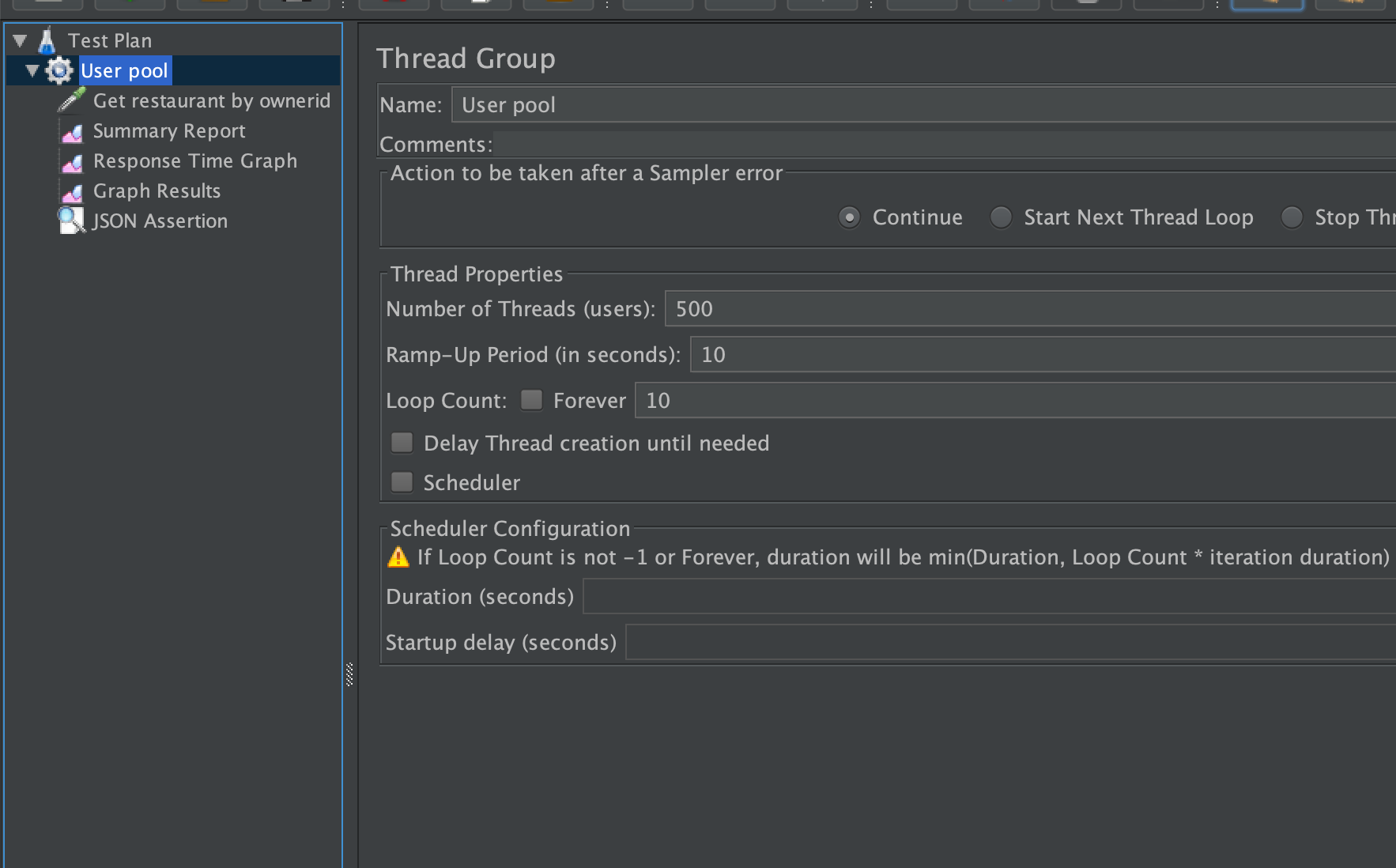
1. 10 calls, 200 concurrent users.

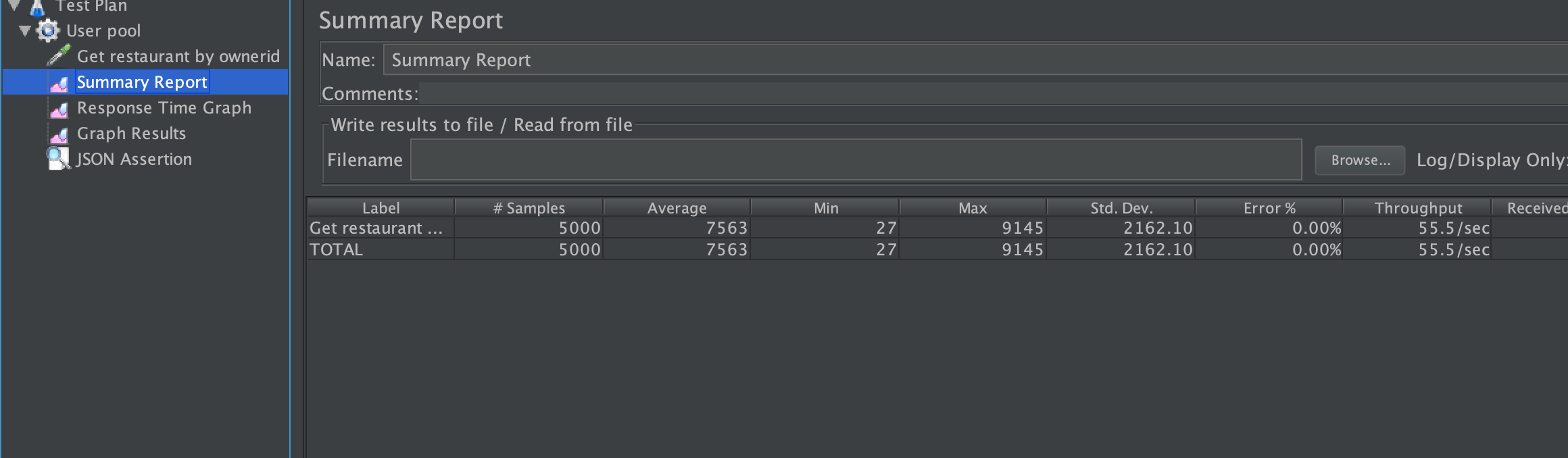


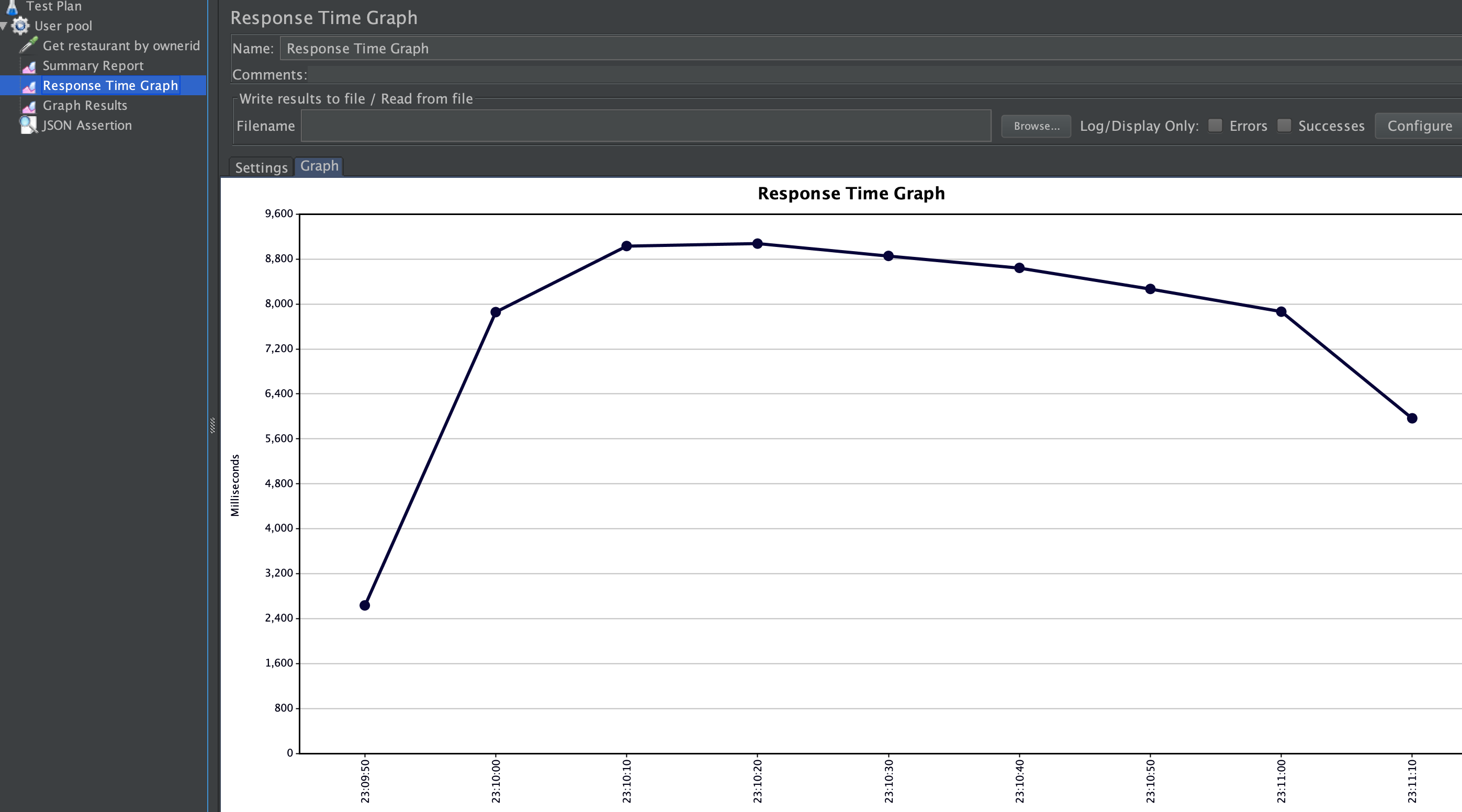




1. 10 calls, 500 concurrent users.

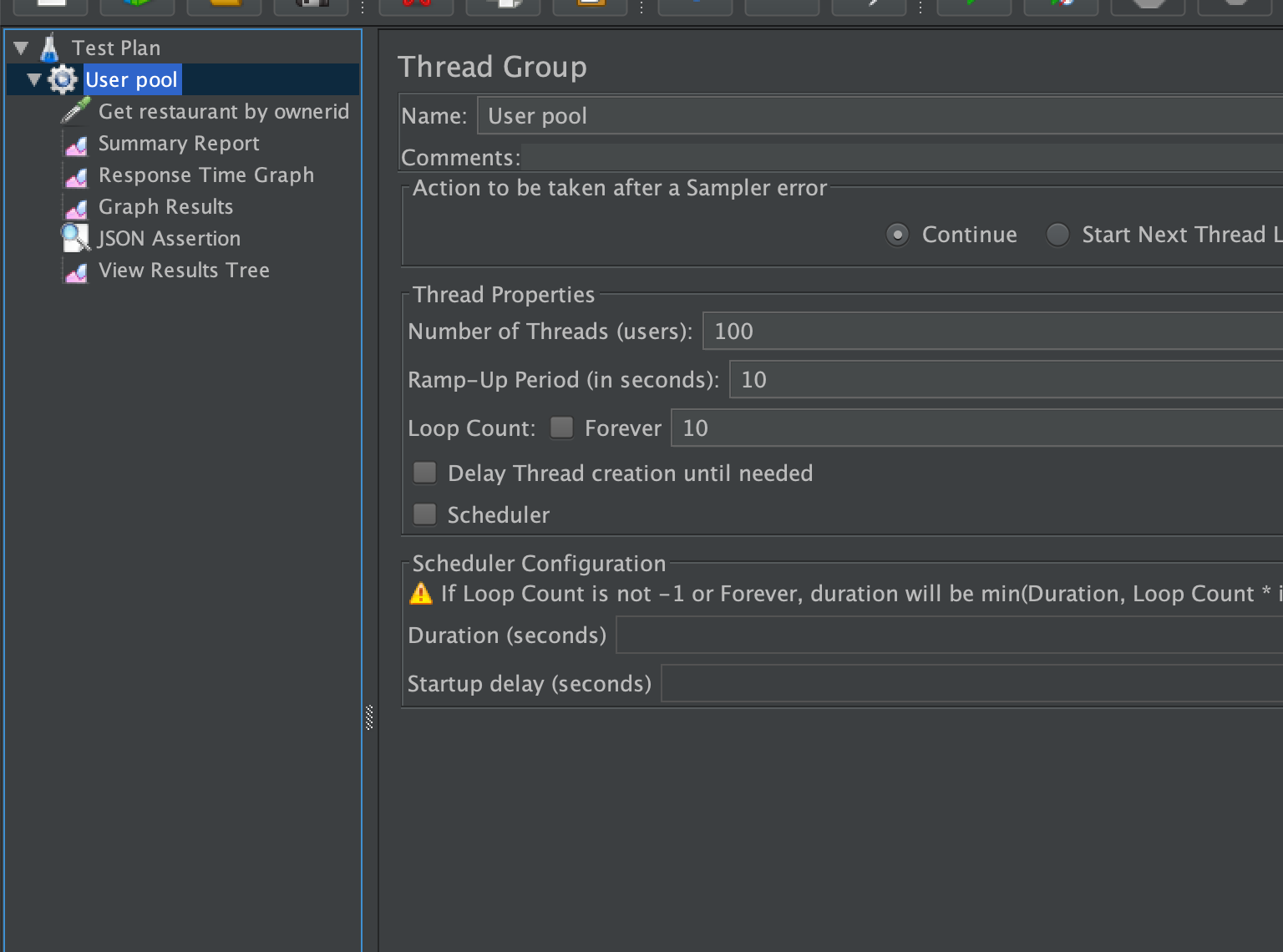


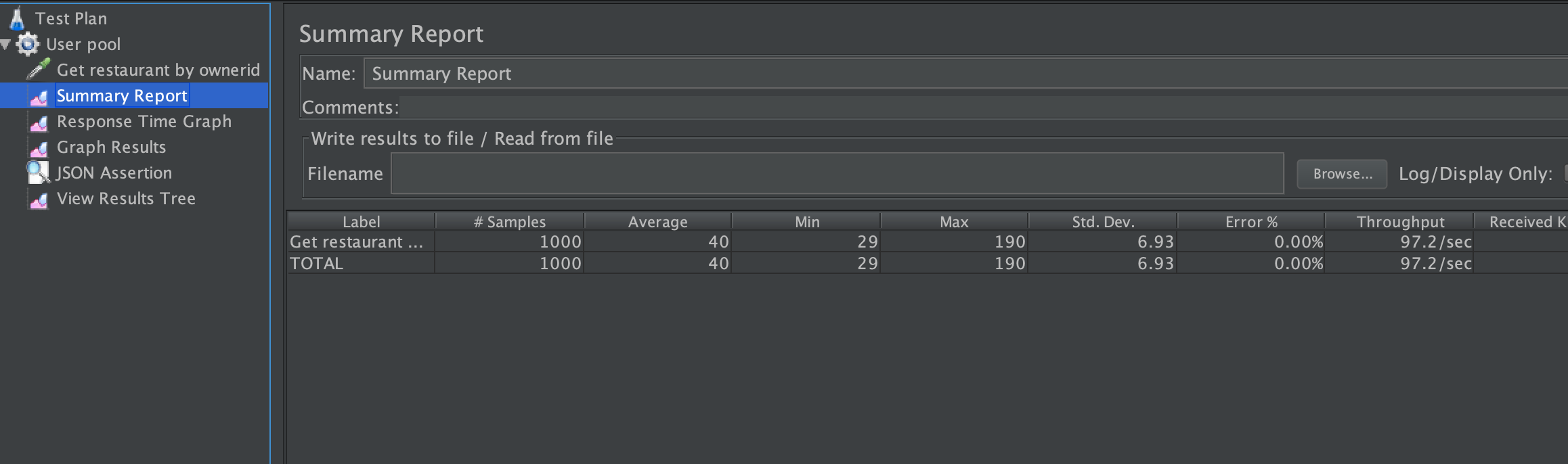


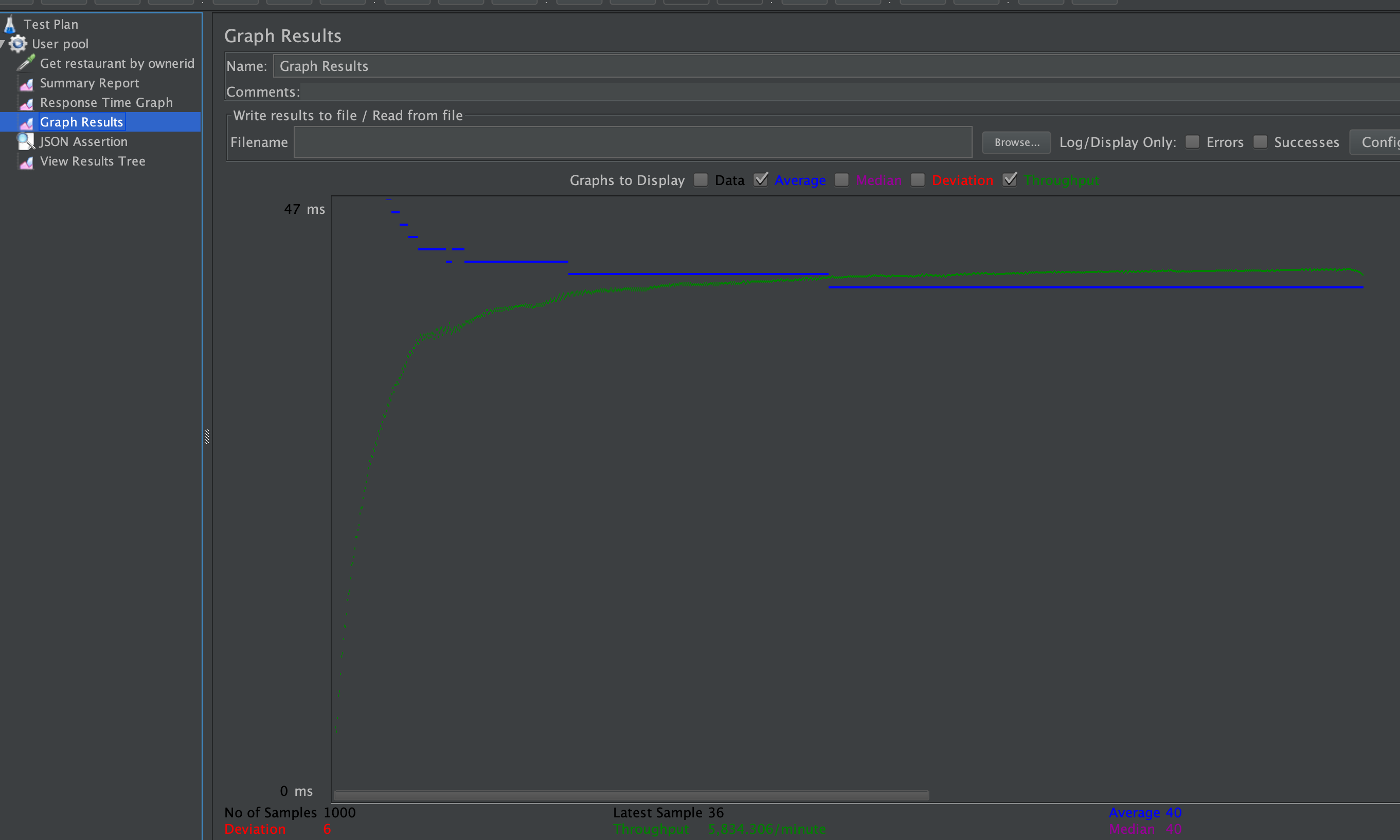


JMeter (With pooling) (50 connections max).

1. 10 calls, 100 concurrent users

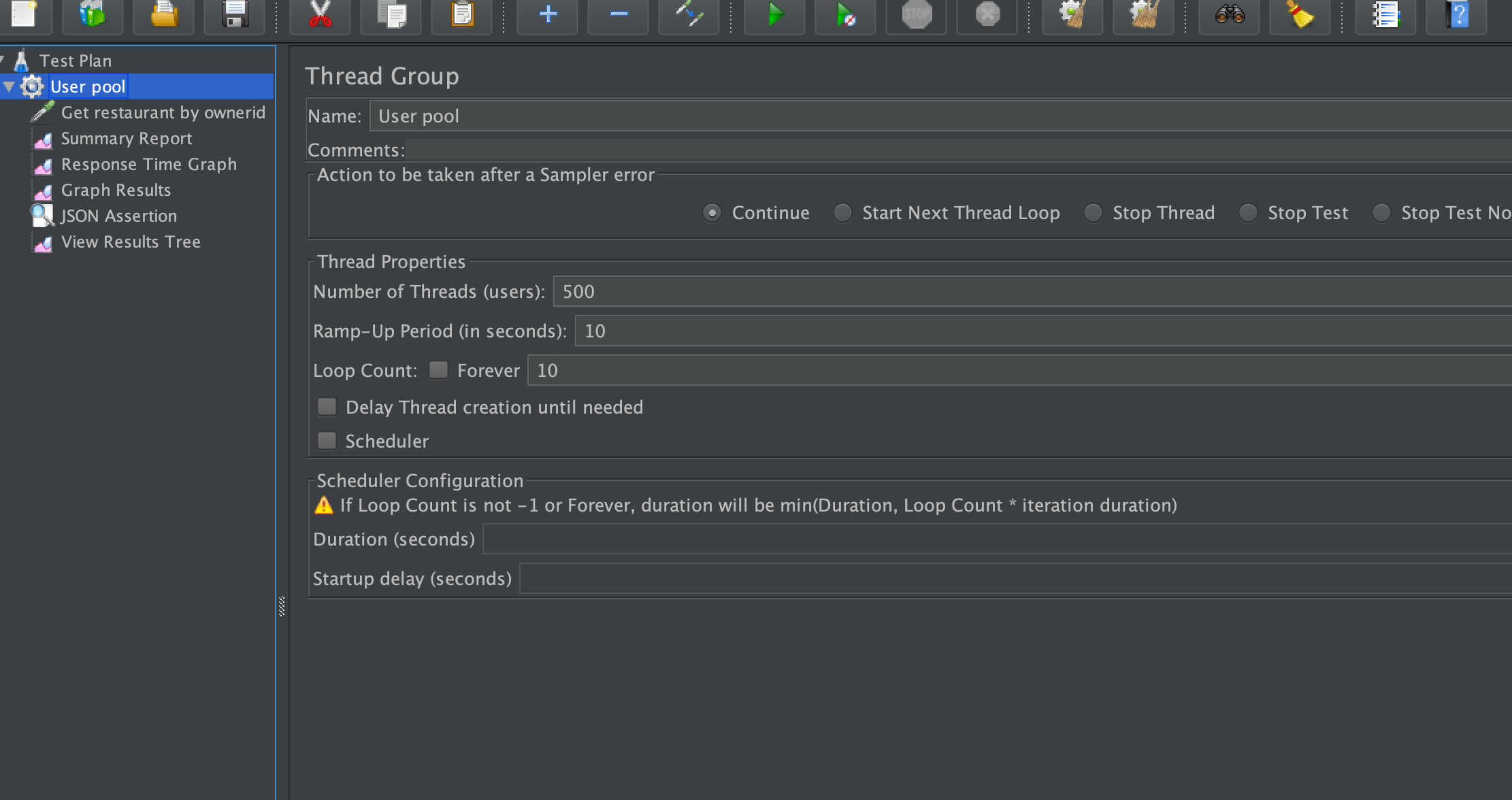


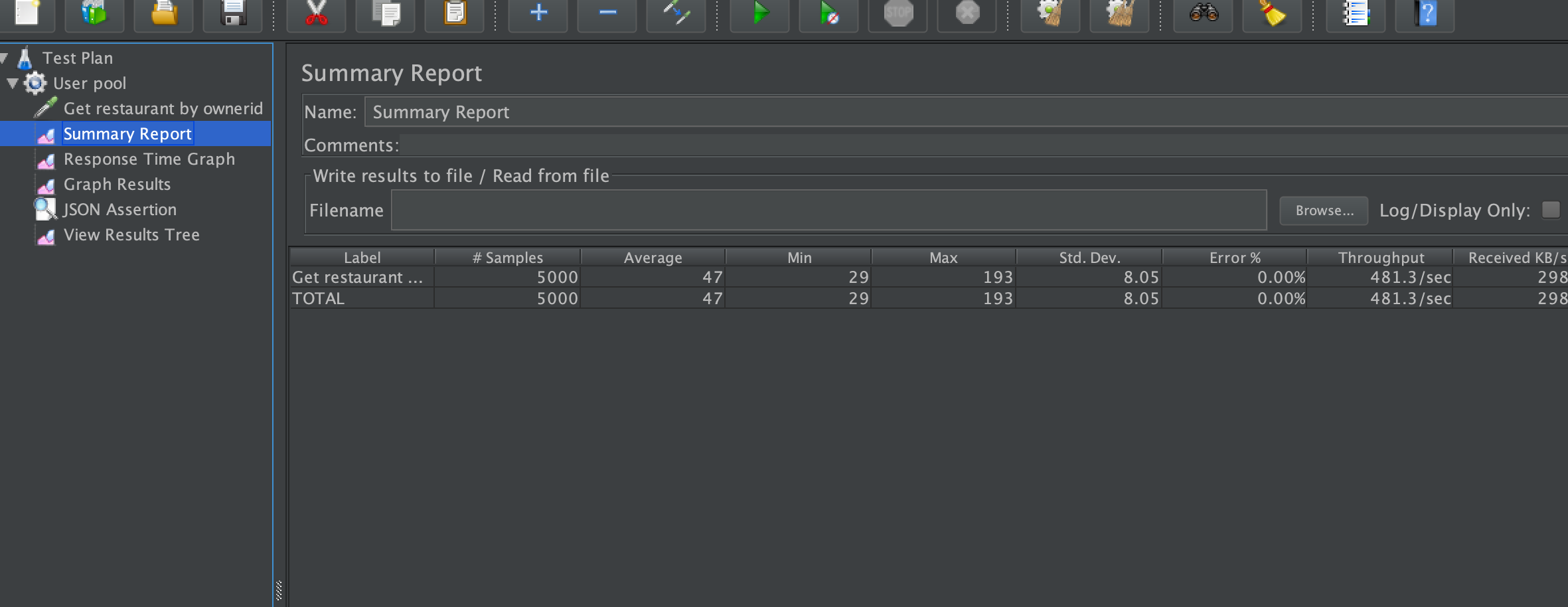


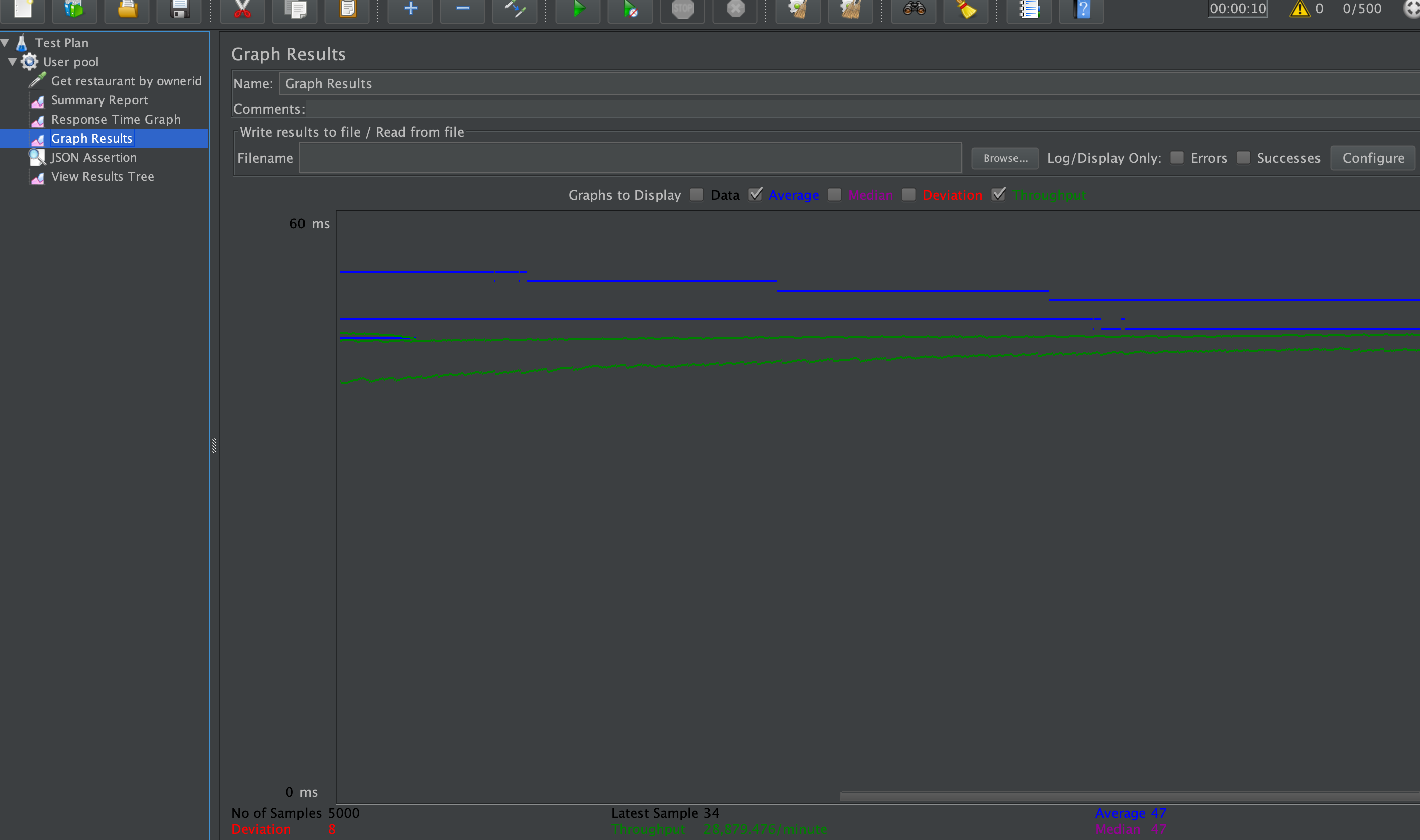


1. 10 calls, 500 concurrent users

We can clearly see the speed improved in servicing the requests by DB with and without pooling.







Mocha testing (Backend)(Unit)

// Mocha is a Node back-end unit test library.

// For any testing library we need a proper assertion library, Node has a basic, Chai is  comprhensive assertion framework

We have selected 5 tasks for mocha unit-testing.

it("Get a restaurant by ownerid", function(done) {

        chai.request('localhost:3001').get('/restaurant/owner/23').end((error, response) => {   // (err: any, res: request.Response) // Definiiton of this function. // Only program/network error will go to error, 200, 404, 500 all are treated as response only, we have to write handler for them.

            if(error) {

                chai.assert.fail(error); // Always fail assertion.

            } else {

                chai.expect(response).to.have.status(200);

                chai.expect(response.body).haveOwnProperty("name");

                chai.expect(response.body).haveOwnProperty("zip");

                chai.expect(response.body).haveOwnProperty("cuisine");

            }

            done(); // to show it's async to chai and not the test case in tht first itself and wait for end-callback.

        });

    });

    it("Get all restaurants", function(done) {

        chai.request('localhost:3001').get('/restaurant/').end((error, response) => {   // (err: any, res: request.Response) // Definiiton of this function. // Only program/network error will go to error, 200, 404, 500 all are treated as response only, we have to write handler for them.

            if(error) {

                chai.assert.fail(error); // Always fail assertion.

            } else {

                chai.expect(response).to.have.status(200);

                chai.assert(Array.isArray(response.body), "Response is not an array");

                chai.assert(response.body[0].hasOwnProperty("name"));

                chai.assert(response.body[0].hasOwnProperty("zip"));

                chai.assert(response.body[0].hasOwnProperty("cuisine"));

            }

            done(); // to show it's async to chai and not the test case in tht first itself and wait for end-callback.

        });

    });

    it("Get a orders by restaurantid", function(done) {

        chai.request('localhost:3001').get('/restaurant/7/orders').end((error, response) => {   // (err: any, res: request.Response) // Definiiton of this function. // Only program/network error will go to error, 200, 404, 500 all are treated as response only, we have to write handler for them.

            if(error) {

                chai.assert.fail(error); // Always fail assertion.

            } else {

                chai.expect(response).to.have.status(200);

                chai.assert(response.body.hasOwnProperty("numberoforders"));

                chai.assert(Array.isArray(response.body.orders));

                chai.assert(response.body.orders[0].hasOwnProperty("cost"));

                chai.assert(response.body.orders[0].hasOwnProperty("status"));

                chai.assert(Array.isArray(response.body.orders[0].menu));

            }

            done(); // to show it's async to chai and not the test case in tht first itself and wait for end-callback.

        });

    });

it("Get a user by id", function(done) {

        chai.request('localhost:3001').get('/user/1').end((error, response) => {   // (err: any, res: request.Response) // Definiiton of this function. // Only program/network error will go to error, 200, 404, 500 all are treated as response only, we have to write handler for them.

            if(error) {

                chai.assert.fail(error); // Always fail assertion.

            } else {

                chai.expect(response).to.have.status(200);

                chai.expect(response.body).haveOwnProperty("name");

                chai.expect(response.body).haveOwnProperty("email");

                chai.expect(response.body).haveOwnProperty("phone");

            }

            done(); // to show it's async to chai and not the test case in tht first itself and wait for end-callback.

        });

    });

    it("Get a orders by userid", function(done) {

        chai.request('localhost:3001').get('/user/1/orders').end((error, response) => {   // (err: any, res: request.Response) // Definiiton of this function. // Only program/network error will go to error, 200, 404, 500 all are treated as response only, we have to write handler for them.

            if(error) {

                chai.assert.fail(error); // Always fail assertion.

            } else {

                chai.expect(response).to.have.status(200);

                chai.assert(response.body.hasOwnProperty("numberoforders"));

                chai.assert(Array.isArray(response.body.orders));

                chai.assert(response.body.orders[0].hasOwnProperty("cost"));

                chai.assert(response.body.orders[0].hasOwnProperty("status"));

                chai.assert(Array.isArray(response.body.orders[0].menu));

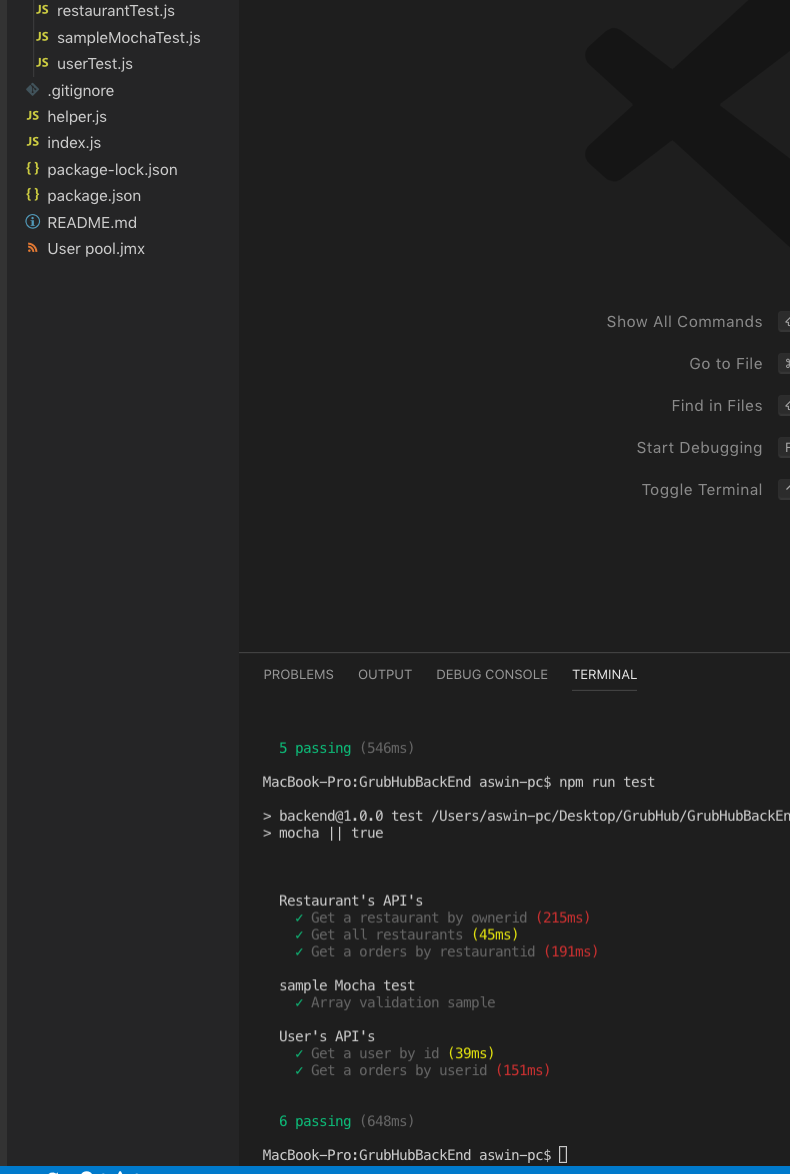
            }

            done(); // to show it's async to chai and not the test case in tht first itself and wait for end-callback.

        });

    });

Mocha testing (Output)



1. Questions.
   1. **Compare the results of graphs with and without in-built MySQL connection pooling of database. Explain the result in detail and describe the connection pooling algorithm if you need to implement connection pooling on your own.**

Answer : From the graph we can clearly see the performance difference between scenarios with and without connection pooling. The average time without pooling is 7000ms whereas with pooling is 45ms.

The thing with pooling is generally a biggest time consumer is DB call, as network calls will happen, authentication and handshake, then query and then data is returned back and the connection is close.

With pooling, we create a pool of DB connections, which are lazily opened when needed and kept in a pool open, they can be use without talking with server again. In this way we have like multi-threaded mechanism to DB. This is very important is modern reactive web-systems where delay should be minimal.

**The states is pooling are connect, acquire and release**.

pool.on('acquire', function(connection) { // Just acquiring a existing conn

   console.log('Connection %d acquired', connection.threadId);

 });

pool.on('release', function (connection) { // relase connection and put in pool

  console.log('Connection %d released', connection.threadId);

 });

pool.on('connection', function (connection) { // this is create a new connection till pool limit when no connection is available.

  console.log('Connection %d connected', connection.threadId);

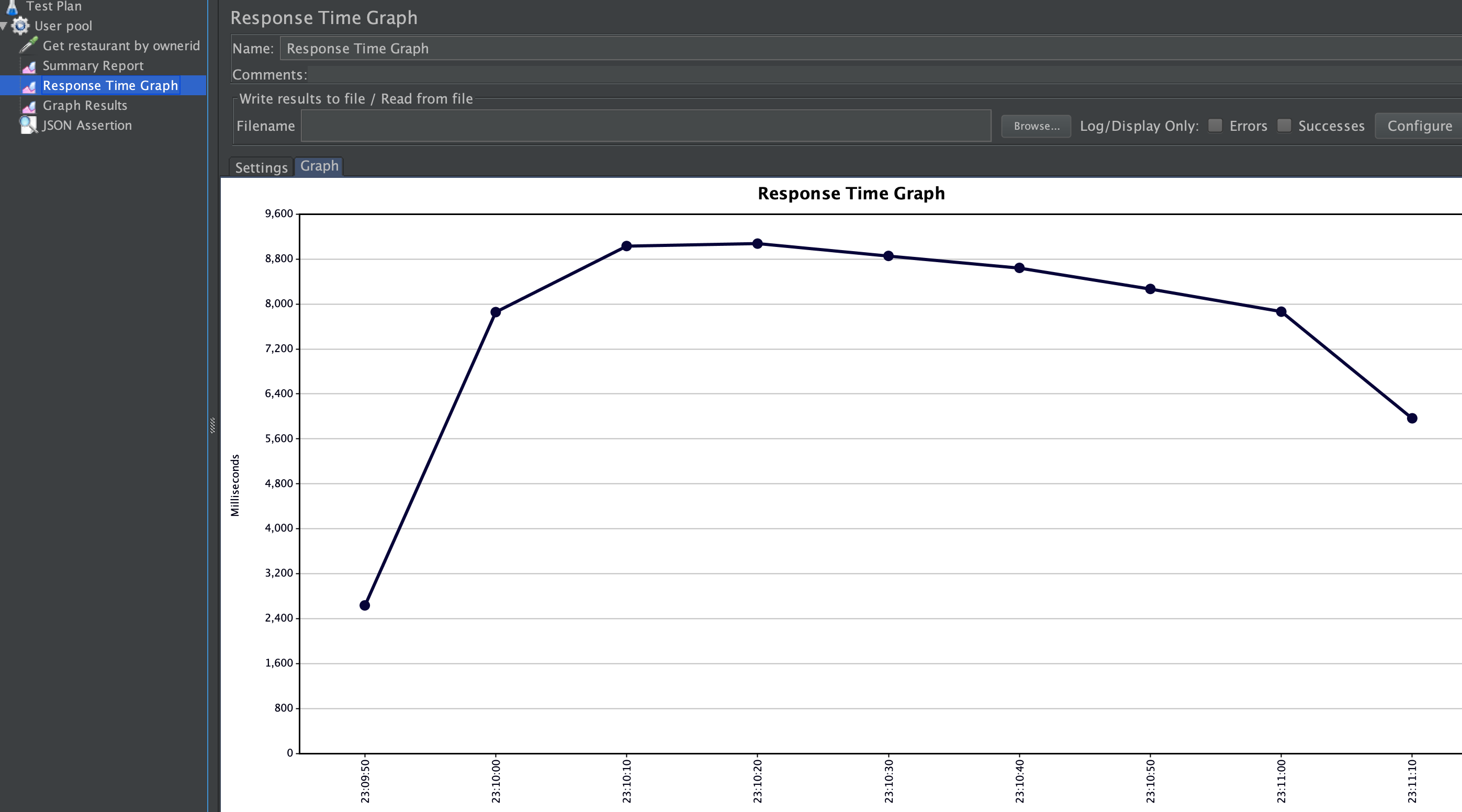
 });

pool.on('enqueue', function () { // Waiting for a connection phase.

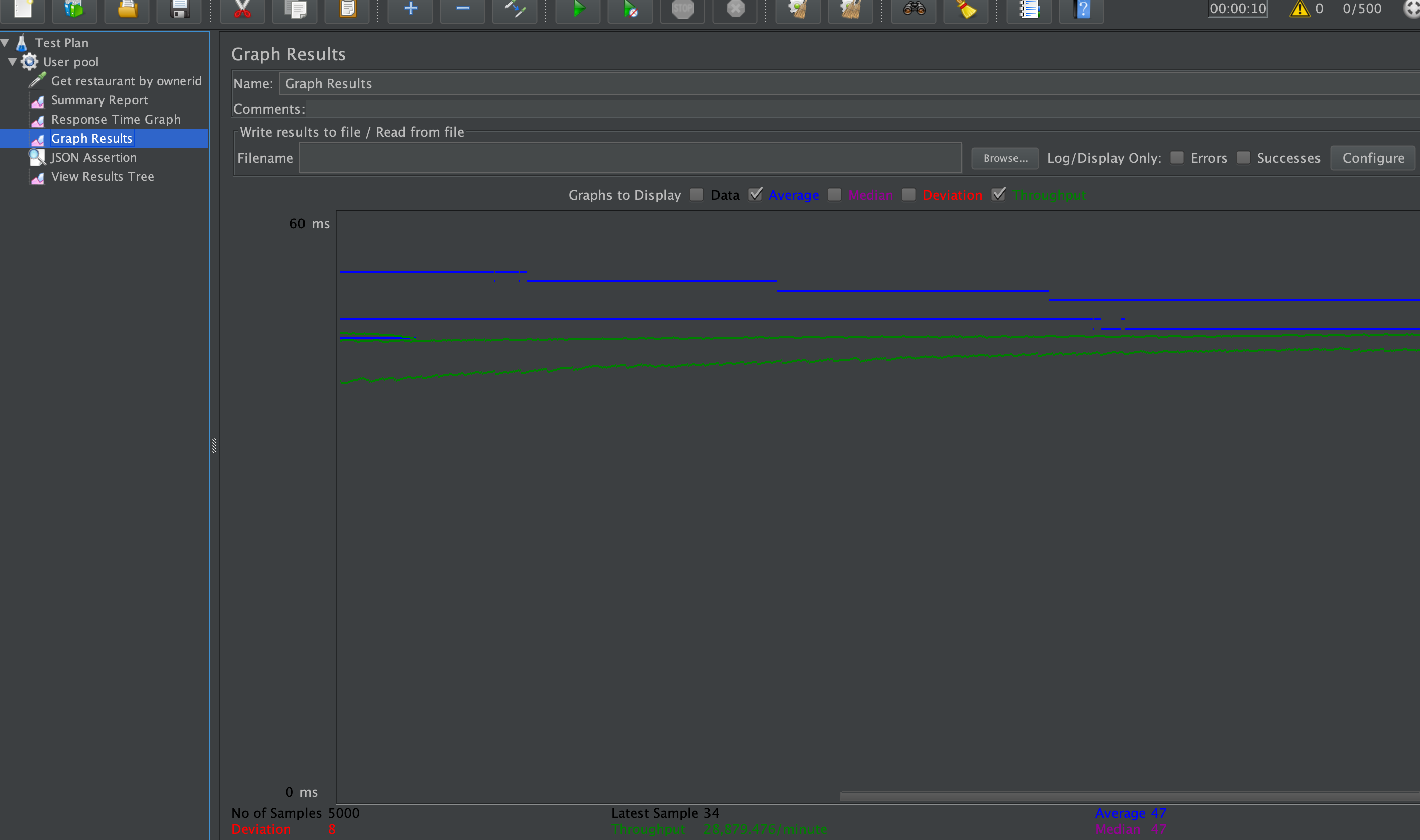
  console.log('Waiting for available connection slot');

});

Without connection pooling.



With connection pooling.



* 1. **What is SQL caching? What all types of SQL caching are available, and which suits your code the most. You don’t need to implement the caching, write pseudocode or explain in detail.**

Caching is generally a process of storing the response and request data of a time consuming process and when there is a identical request, instead of doing a time intensive process, we return data from the cache.

SQL caching is implemented in many Databases in multiple ways specific to themselves, as we are using MySQL system here, MySQL does the caching for SELECT queries alone.

When caching mechanism is implemented and identical query(Not even extra comments are consider identical) after a similar SELECT query and the underlying table is not modified in mean-time, instead of querying again, data from cache is retrieved thus saving time.

Consider in our example which we retrieve a restaurant’s menu, instead of getting the menu from DB, unless the table is changed, we can get the data from cache. This is one place we can implement cache in our system.

Note : We are using Redux, generally to reduce DB calls.

**SELECT SQL\_CACHE \* FROM menu where restaurantid=7;**

Is the query, when we run this first time, data is retrieved and stored in cache, when we call this again, data is retrieved from cache instead of Database.

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

Yes, the current strategy I am using is scalable, In our system, we are storing cookies in user’s end system and we are doing the verification and page authentications in front-end itself. This has some big downfalls, anyone can interpret the cookie data when it is transmitted and re-produce it. But this method can handle horizontal session scaling.

In typical system, the back-end sets a cookie with session id and when the user requests, it send the session id, based on session id , the user is authenticated from backend and can be allowed to access resources, the thing is here we need to have a centralised mechanism to do this session verification as there will be many systems serving out customers.