**CMPE 273 – Lab 2**

**Sricheta Ruj (SJSU ID - 012527606)**

**Goal:** To Develop Freelancer like scalable web application developed in MERN Stack and using Kafka-zookeeper for managing messaging queue.

**Purpose of the system**: The purpose of the system is to develop a real-like Freelancer web application. The application helps professionals to get their work done by engineers simply by sitting at home. It has following features:

**Features included:**

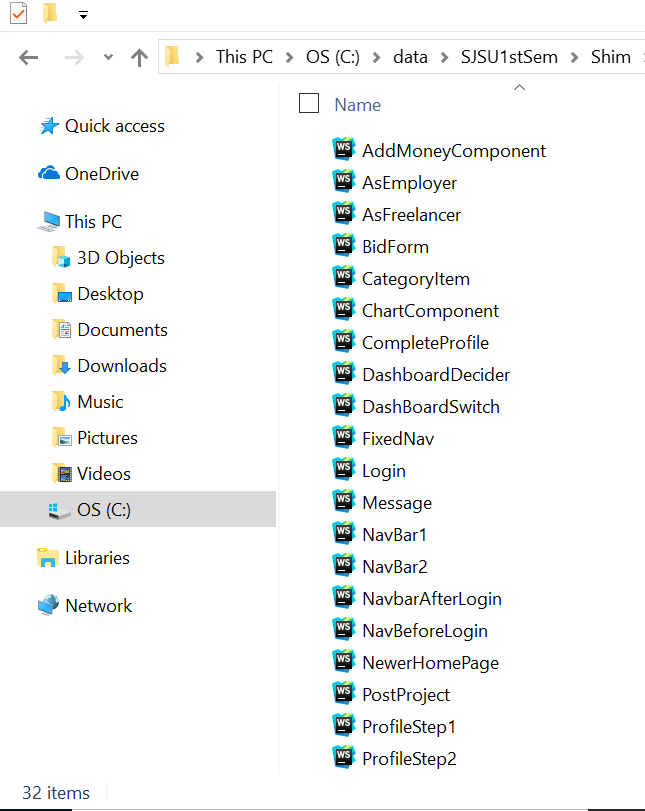
1. Signup as a Freelancer and as a Employer
2. Login
3. Set up profile
4. Dashboard with pagination and filter on Open, Closed or Hiring status (for Employer a well as Freelancer)
5. Home page View all open Projects (with search by project name and skill)
6. Post a Project (upload file)
7. Bid for a project
8. Hire a freelancer
9. Once hired, send an email.

**Technology Used**:

**Client side:**

1. **React** as View in MVC architecture used reusable Components
2. **React-Bootstrap-Typeahead** to choose multiple skills
3. **Redux** for state maintenance.
4. Used **react-file-download** to interpret data stream.
5. **axios** to call the REST service.
6. Used **react-router-dom** to handle routing between different pages.
7. **React-StepZilla** to show progress view of complete profile page.
8. Used “**redux-thunk**” as middleware to support data flow between redux store and react components.
9. **React-chartjs** for showing graphs.

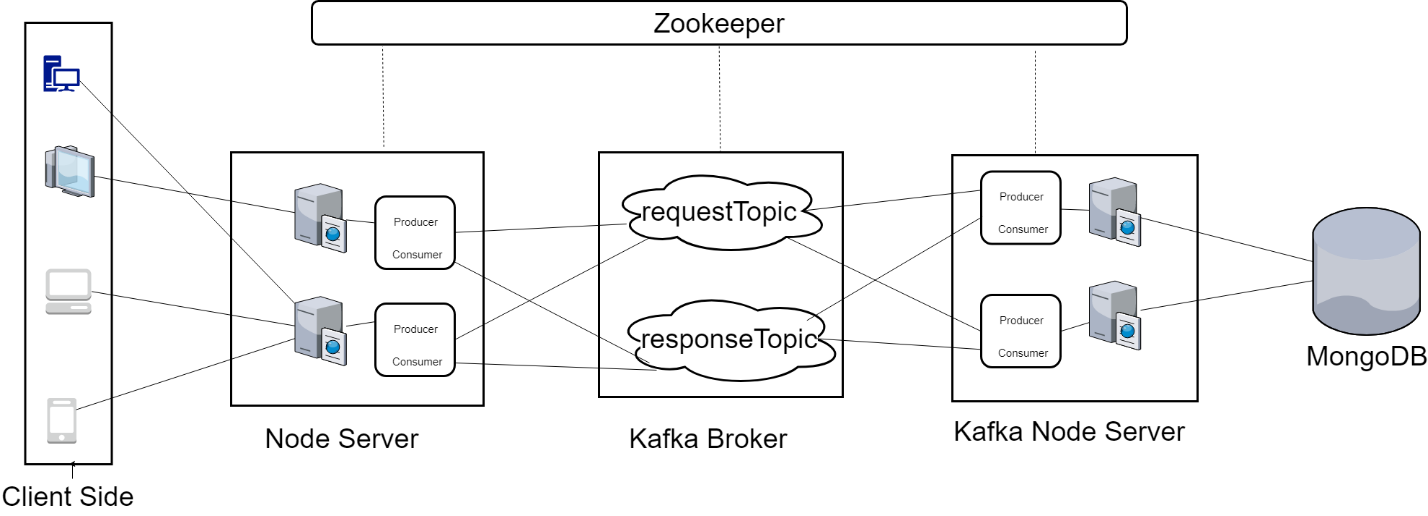
**Total of 32 Components Used :**



**Server side:**

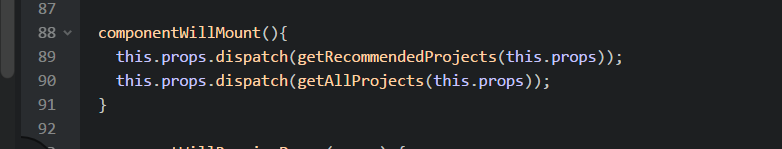
1. Used **Mocha**, **Chai** and Supertest for unit testing of rest api’s
2. **Express** as web application framework.
3. Used **Apache JMeter** for load testing
4. **Passport** for authentication and session management.
5. used **fs-extra** to read the file and send as a blob to client side to display image
6. Used **JWT** for session maintenance.
7. **Nodemailer** for sending e-mails.

**Architecture Diagram** (made using draw.io)

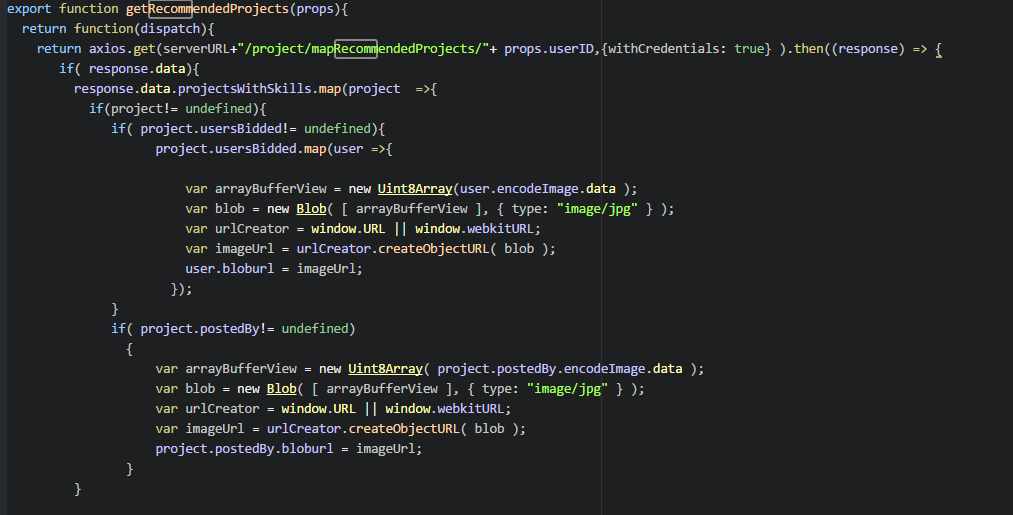


**The flow of searching of relevant projects can be explained as below:**

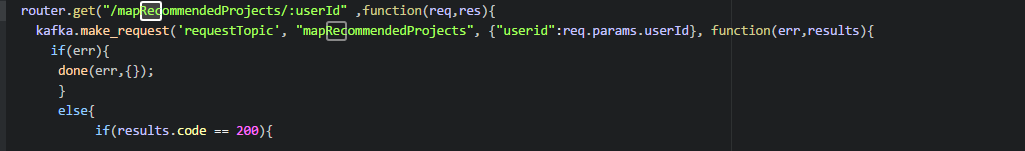
1. After log in to the application, user id get stored in the localStorage.
2. In React lifecycle method componentWillMount, action name “GET\_RECOMMENDED\_PROJECTS” is dispatched passing the userid as argument.



1. A REST API is called to the Node Server



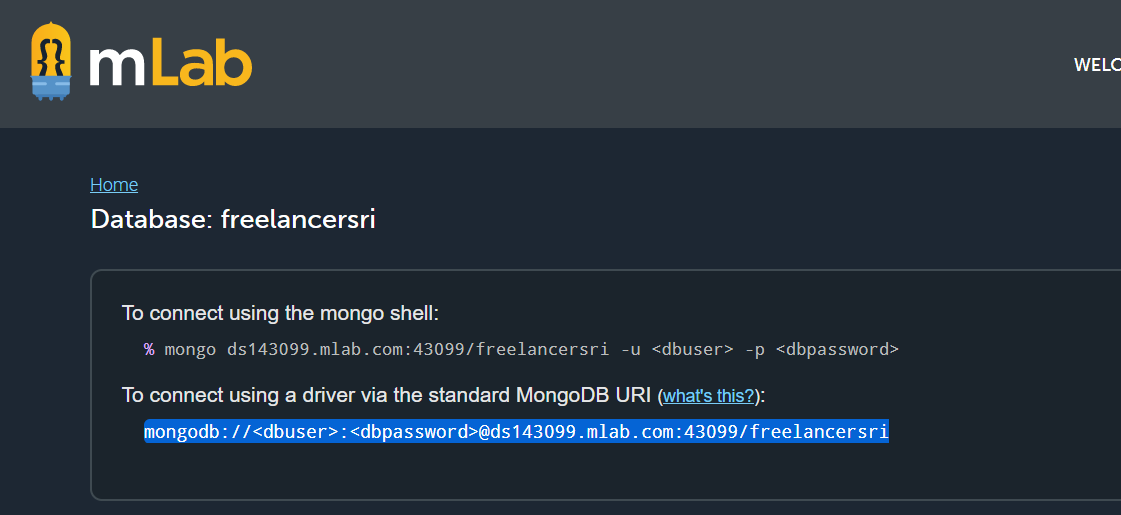
1. At Node server, the userid is sent to requestTopic as messages using Producer.

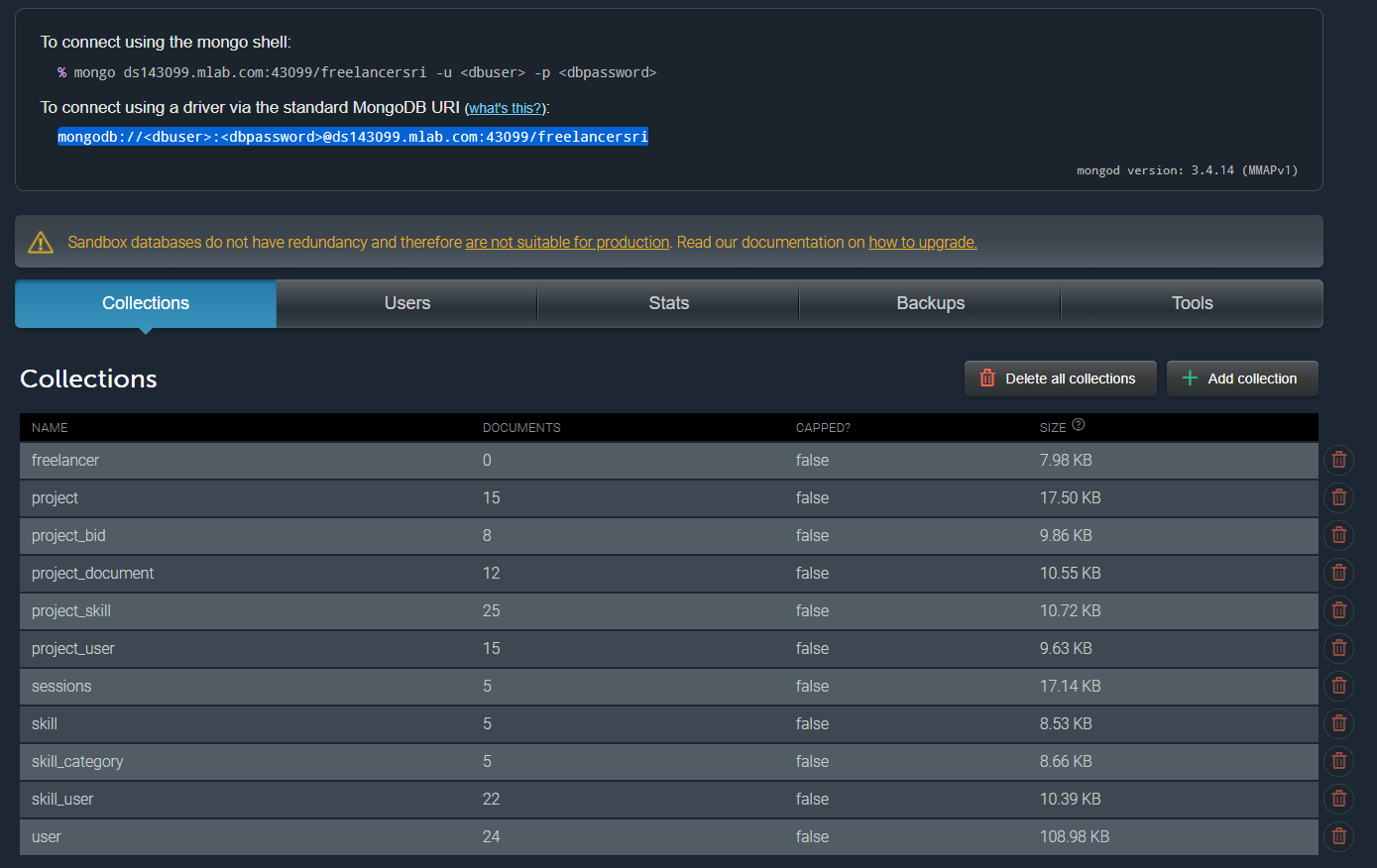


1. Kafka Broker and Zookeeper manage the messaging queue. The messages which are published at requestTopic will be read by consumer of the Kakfa Node sever.
2. Mongo DB queries will be executed at Kafka Node server
3. Messages will be sent back to Node server using responseTopic.

**Database design**

**Database used:** Mongo DB(for development) and MLab (for production)



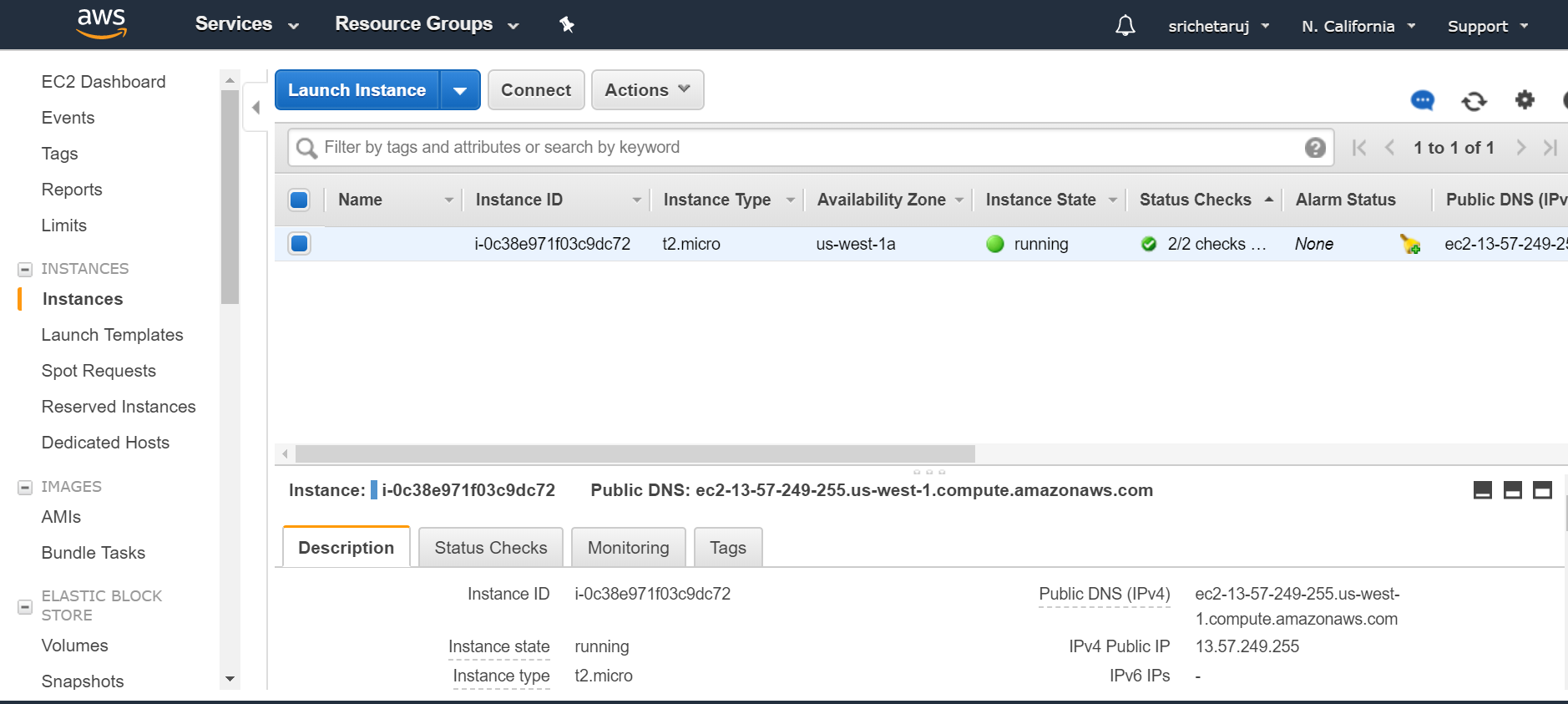


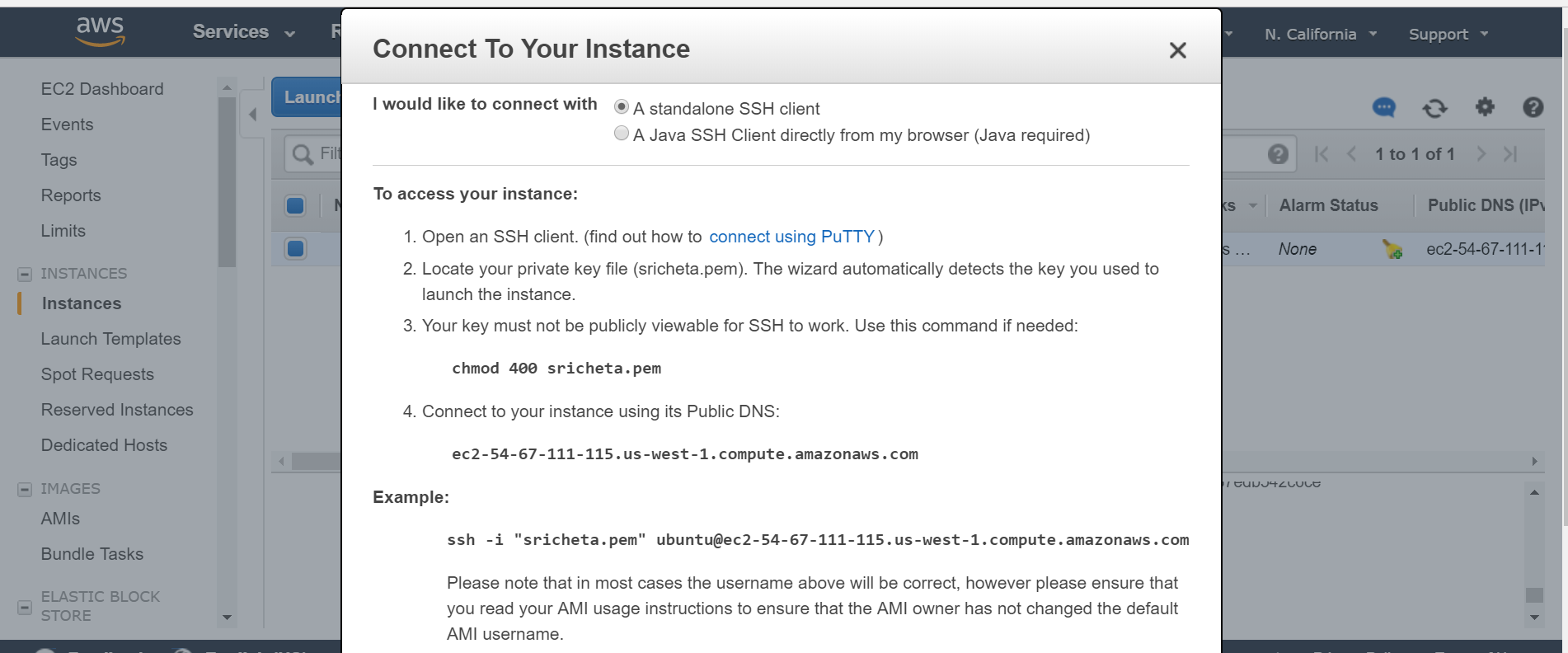
**Hosting:**

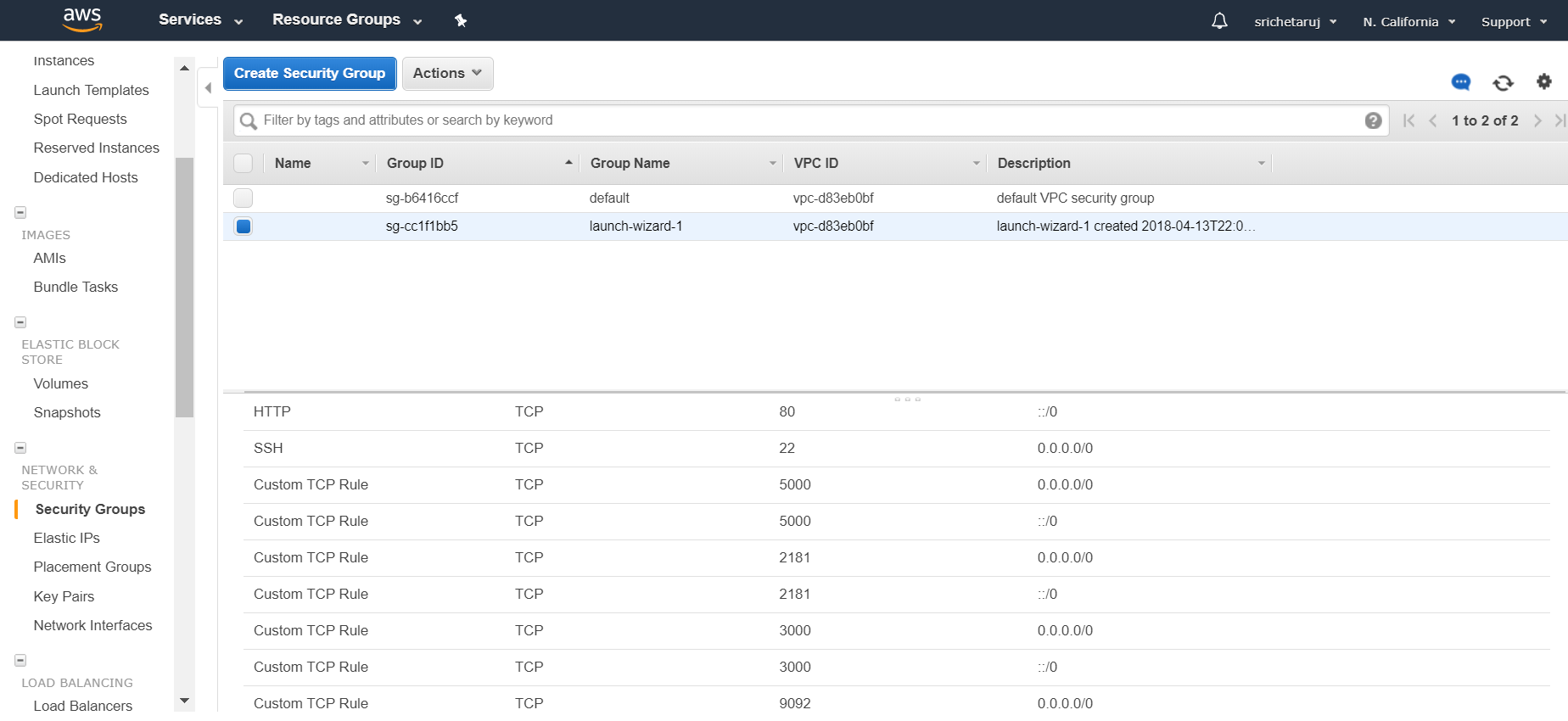
* Hosted on AWS EC2 Linux instance (10 GB)

**Steps followed:**

1. Created a new EC2 instance on AWS as per instructions which are quite self explanatory
2. Connected using SSH command as given in the below screenshot.
3. Push my code to github.
4. SSHed to the instance and did git pull.
5. Installed Kafka and Zookeeper on Linux machine of EC2 instance
6. Started the client, Kafka node server and server as I do in local machine

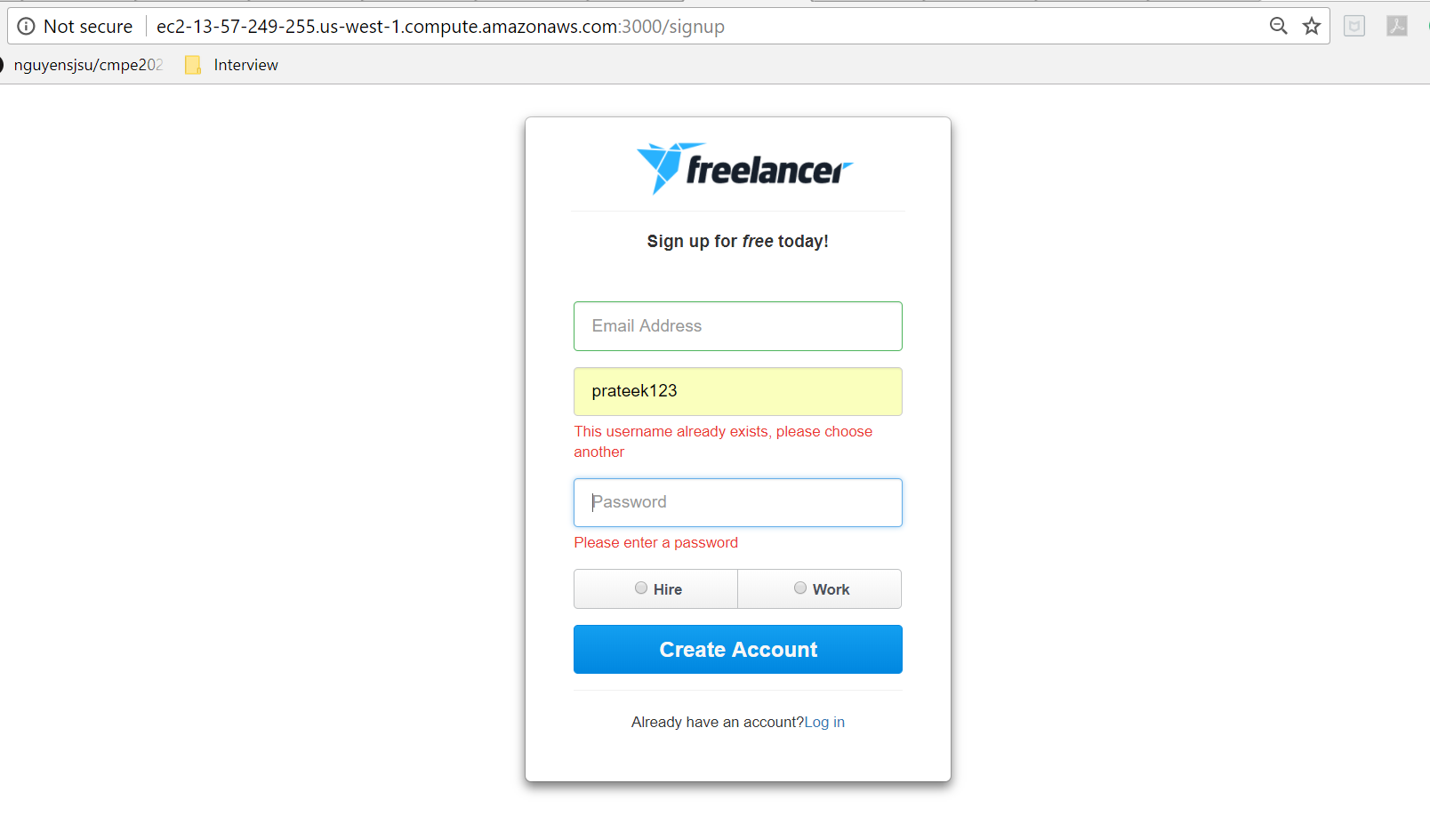




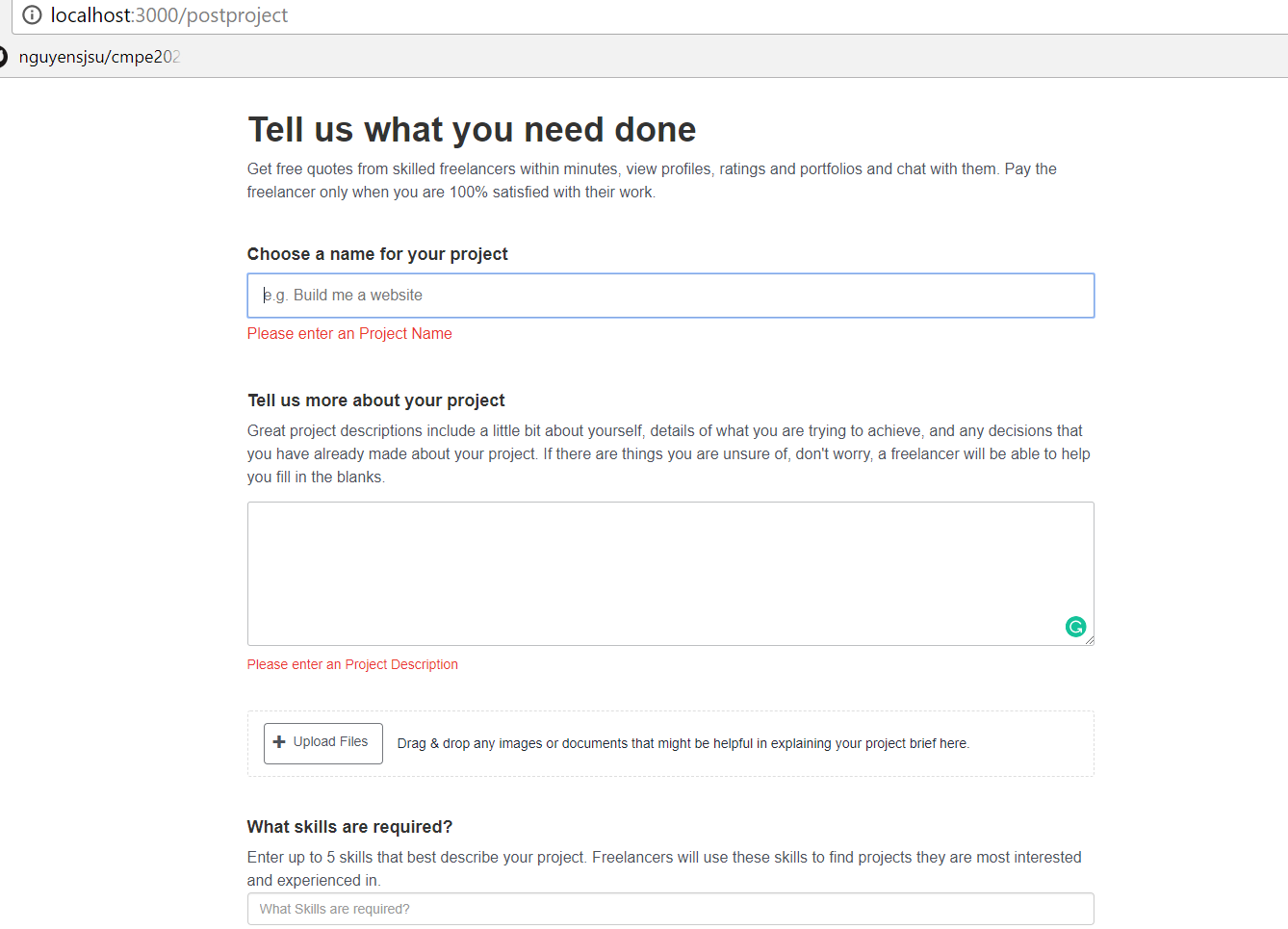


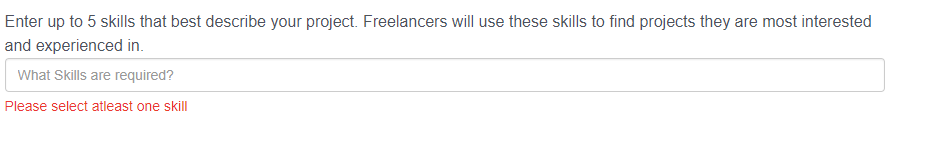
**Functionality**

1. **Signup**

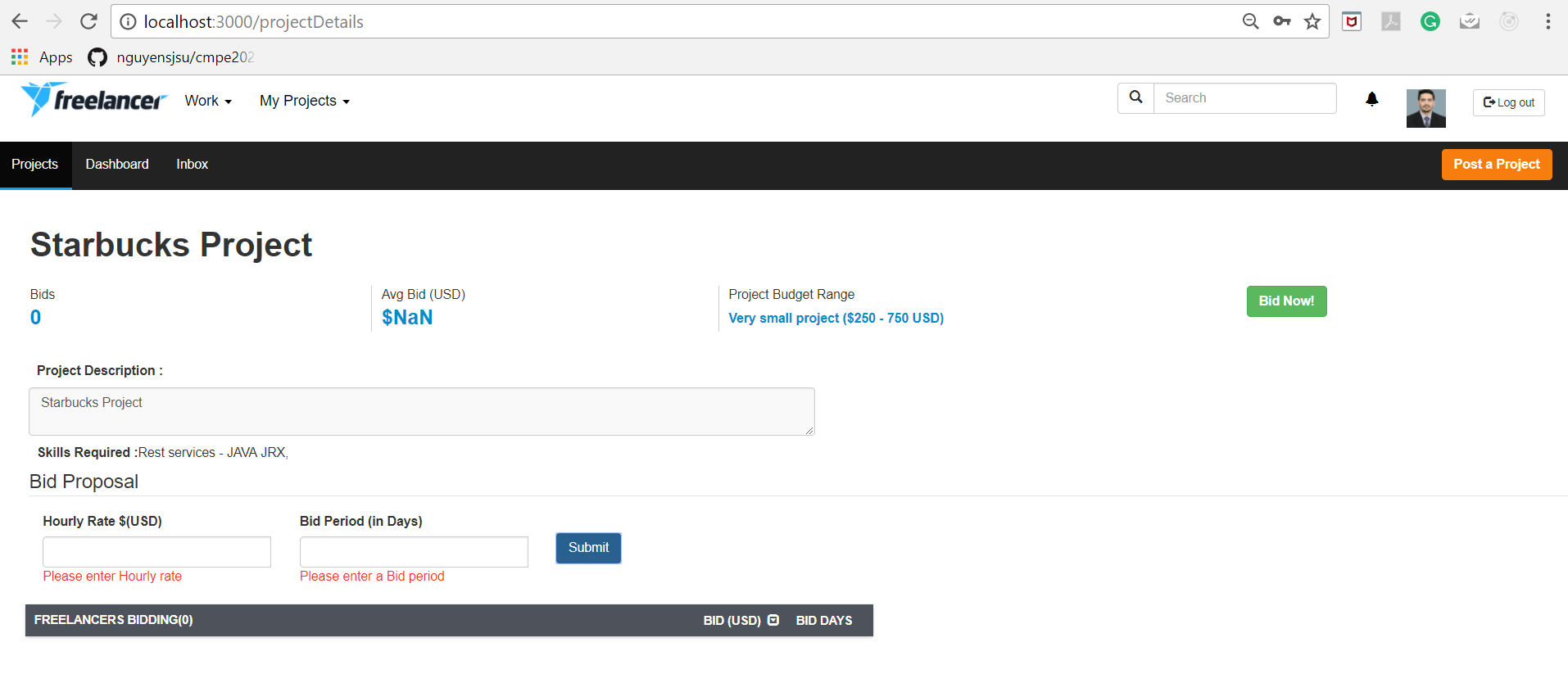


1. **Post Project validations**

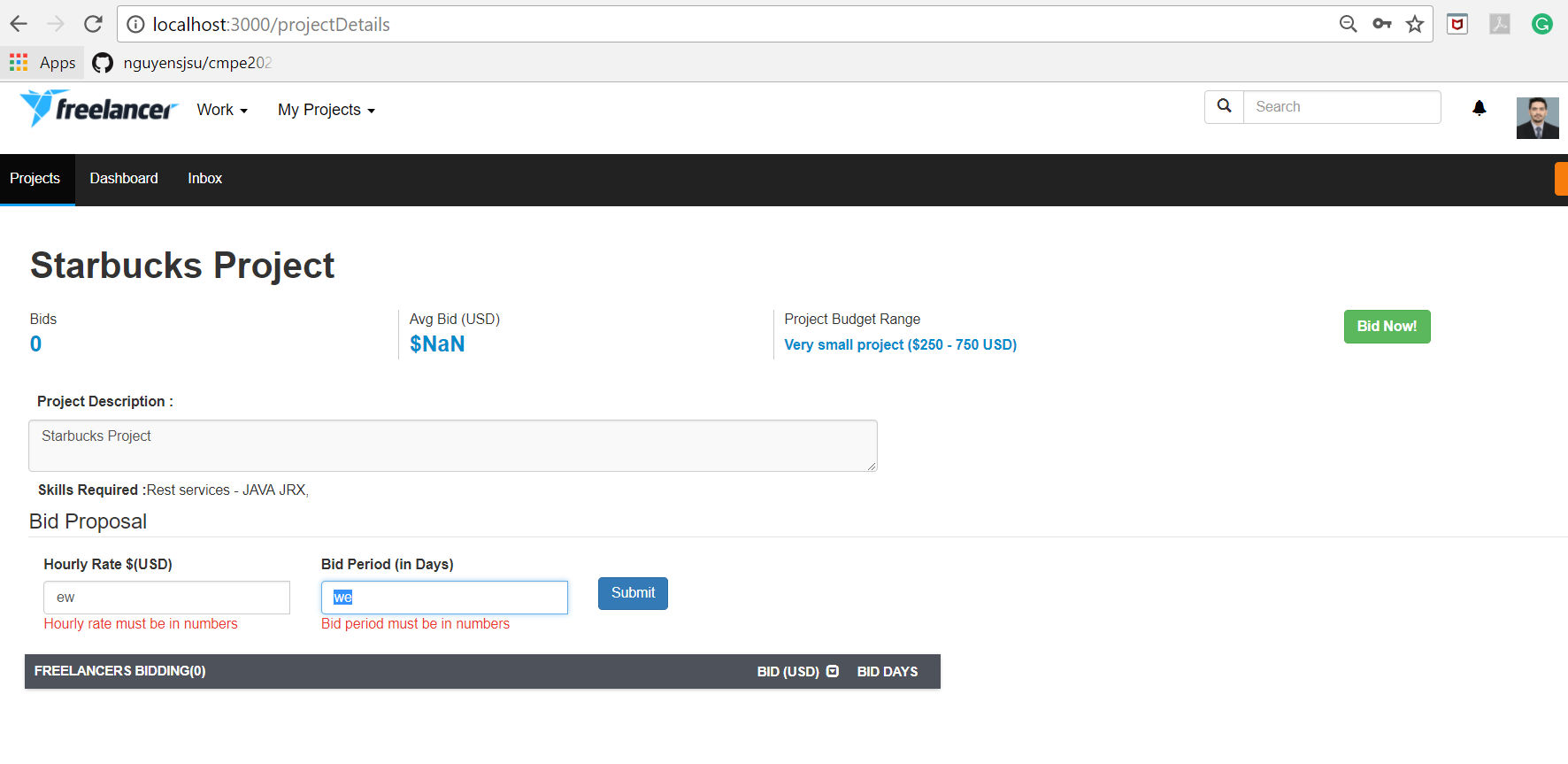




1. **Bid Form Validations ( check whether value has been entered)**

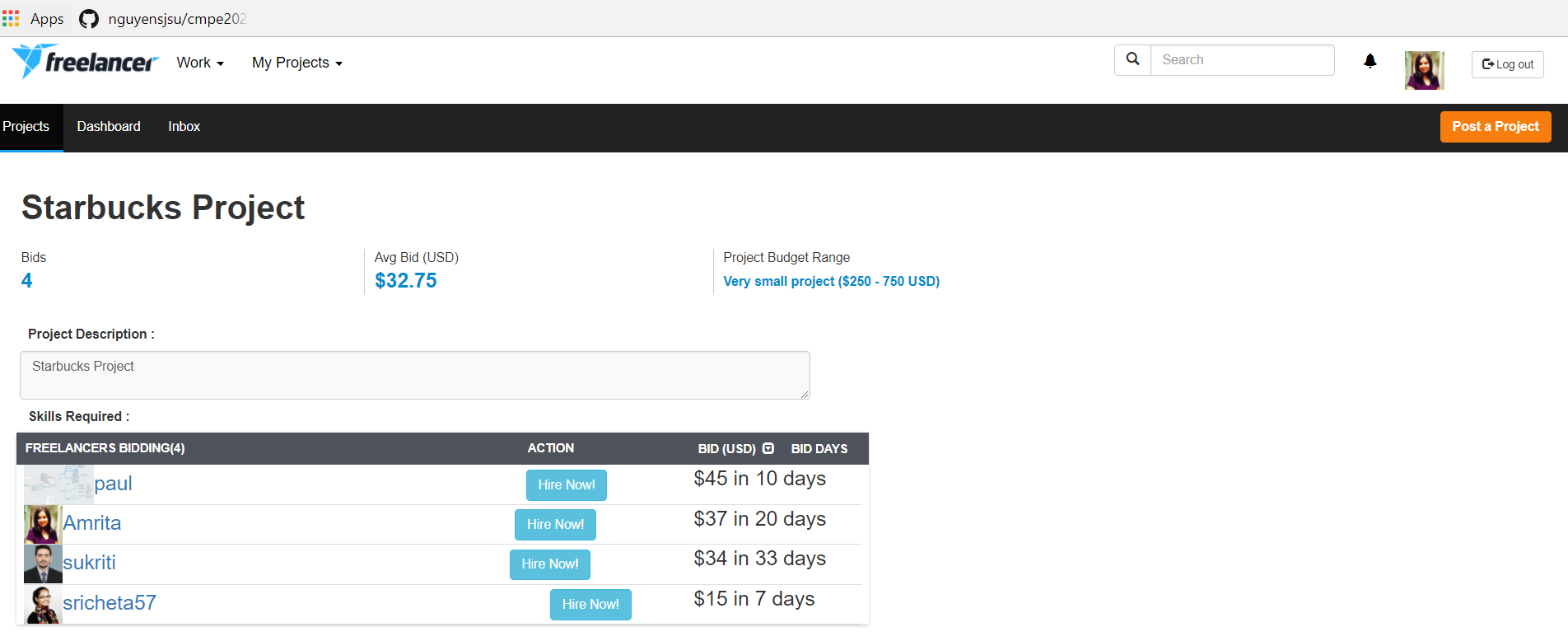


1. **Bid Form Validations ( check whether value is numeric or not)**

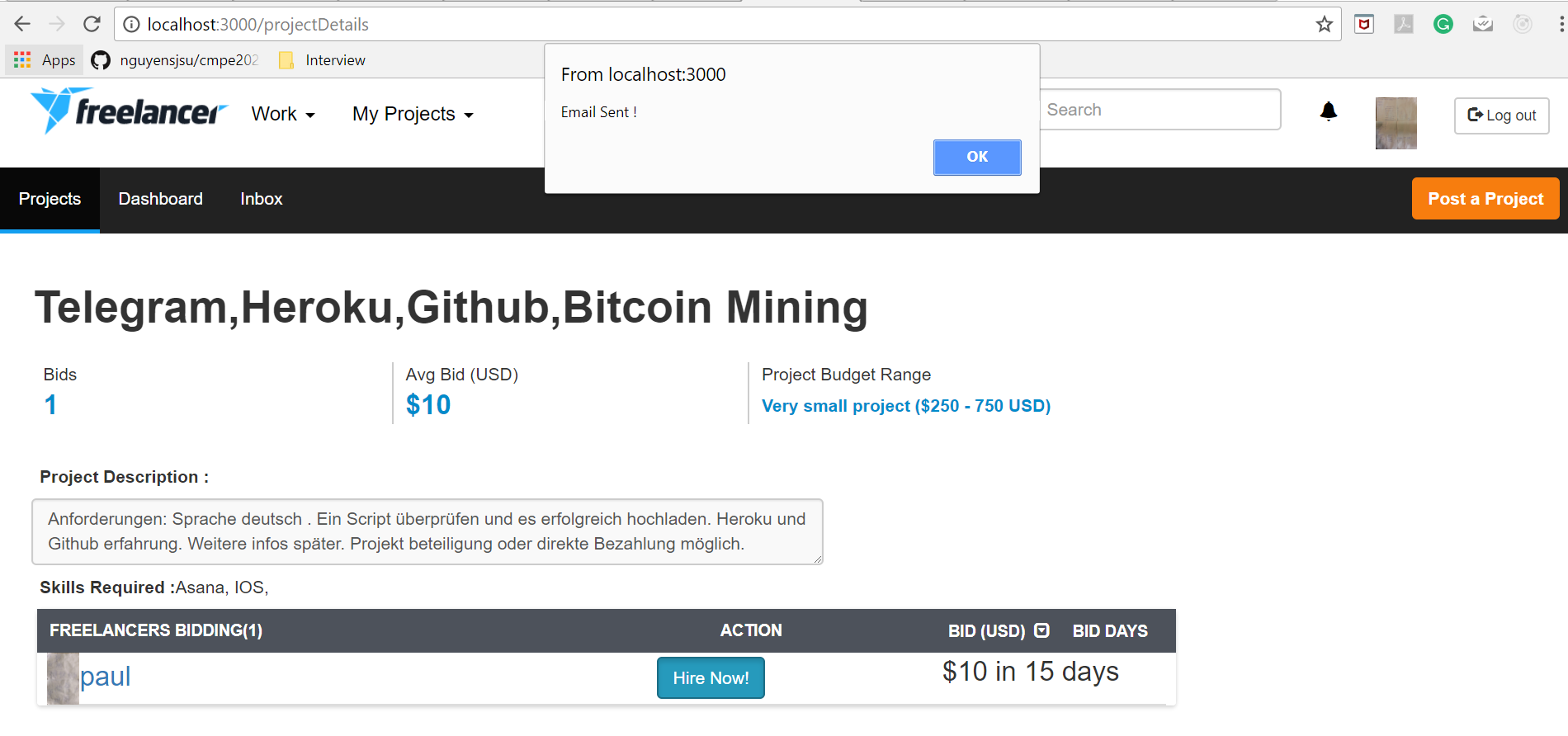


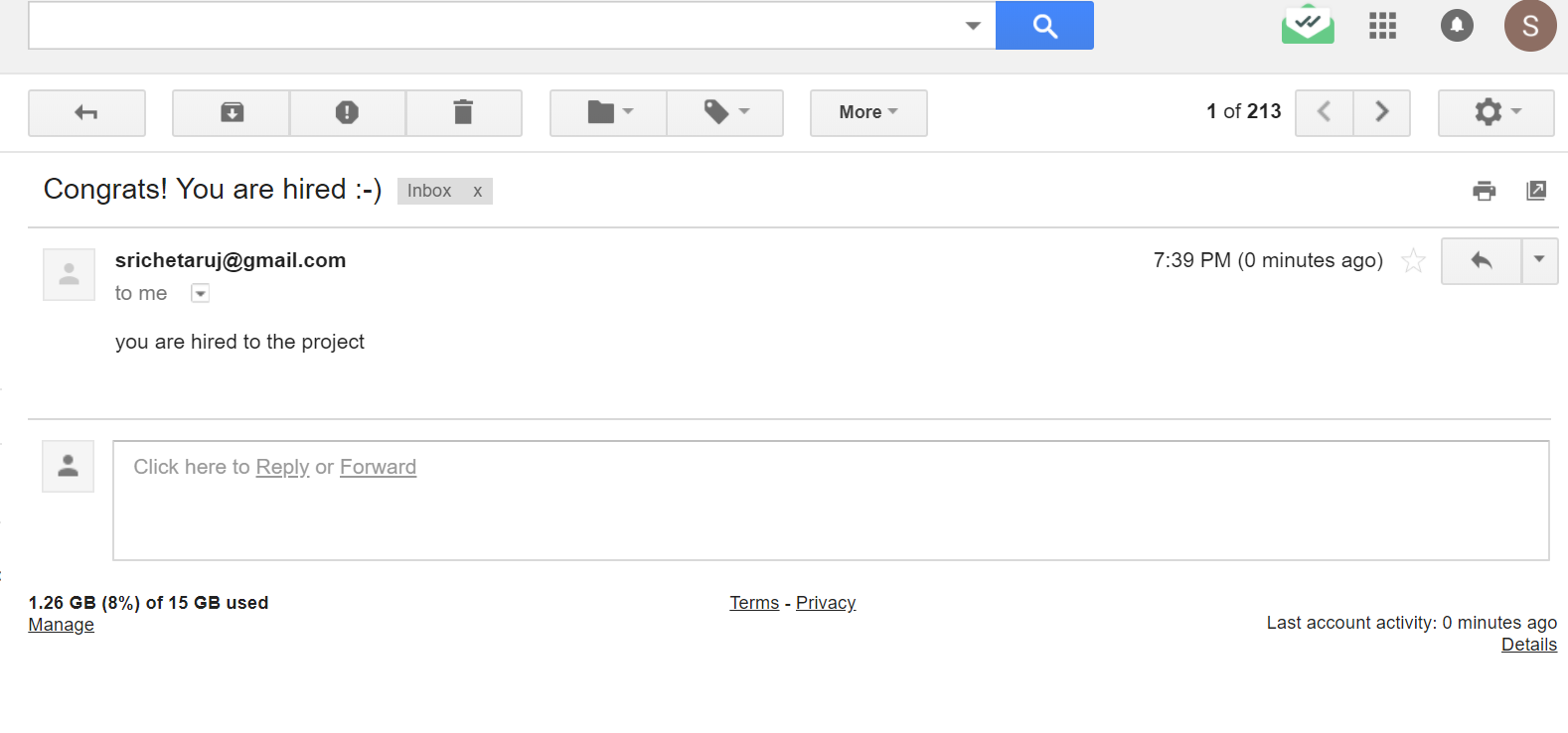
1. **Sorting on user bid list**



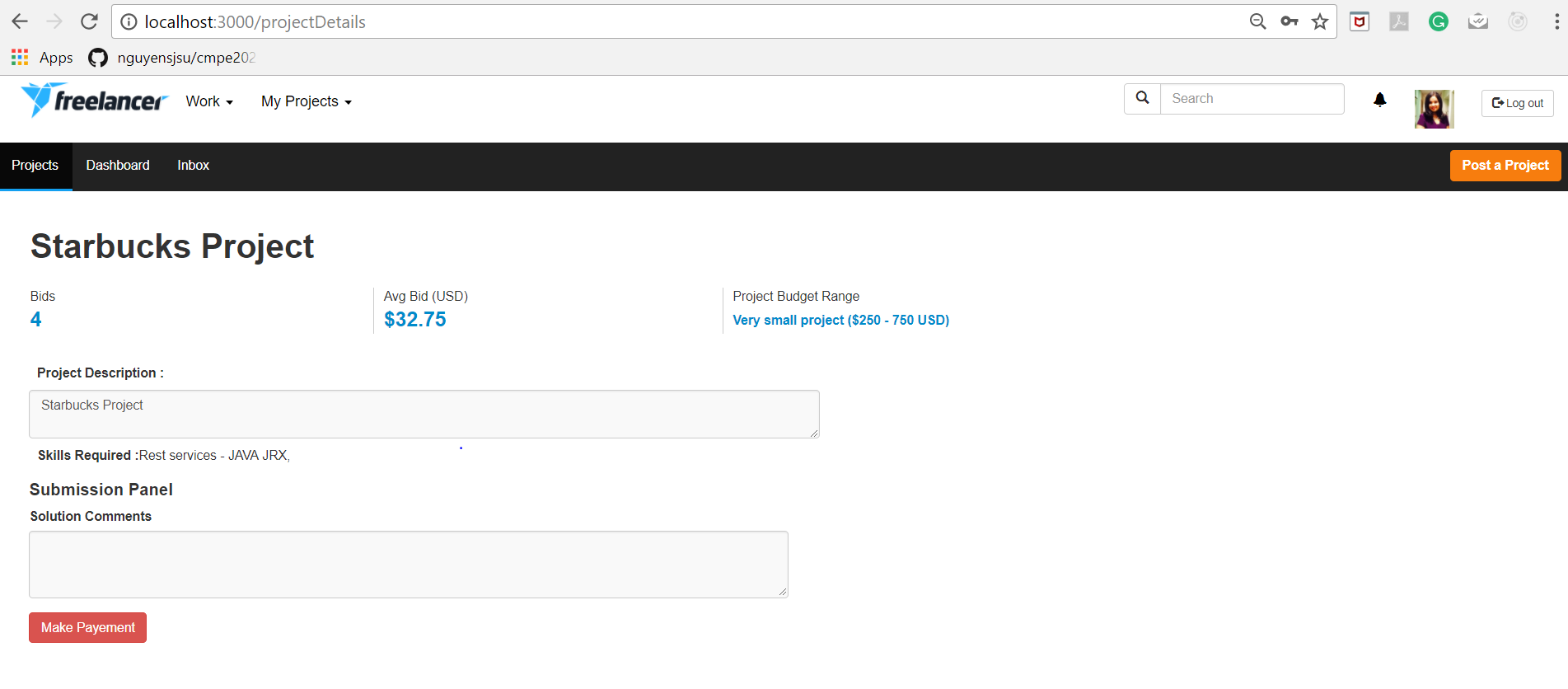


1. **After click on hire now button, Email is sent to the hired freelancer!**

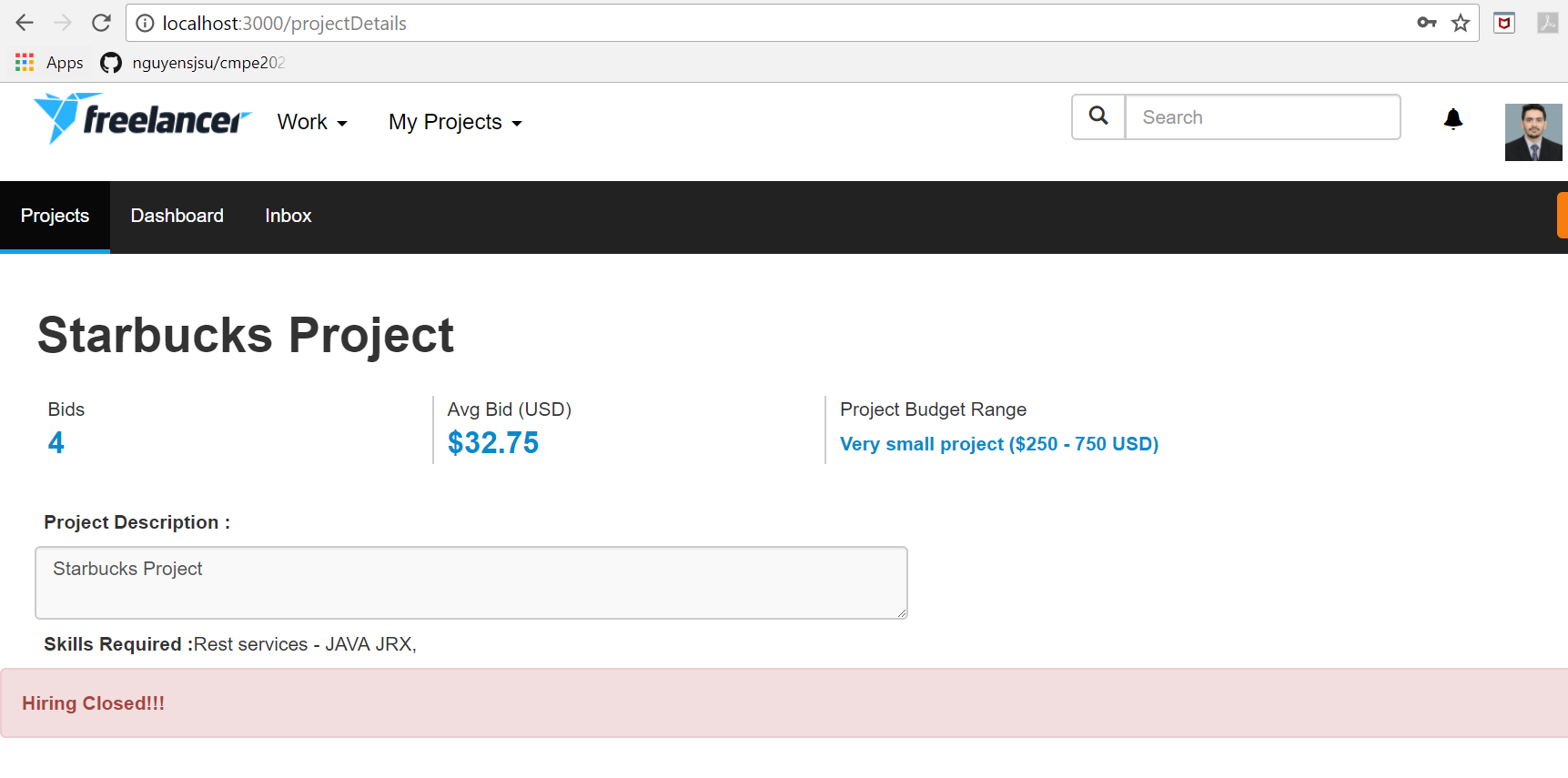




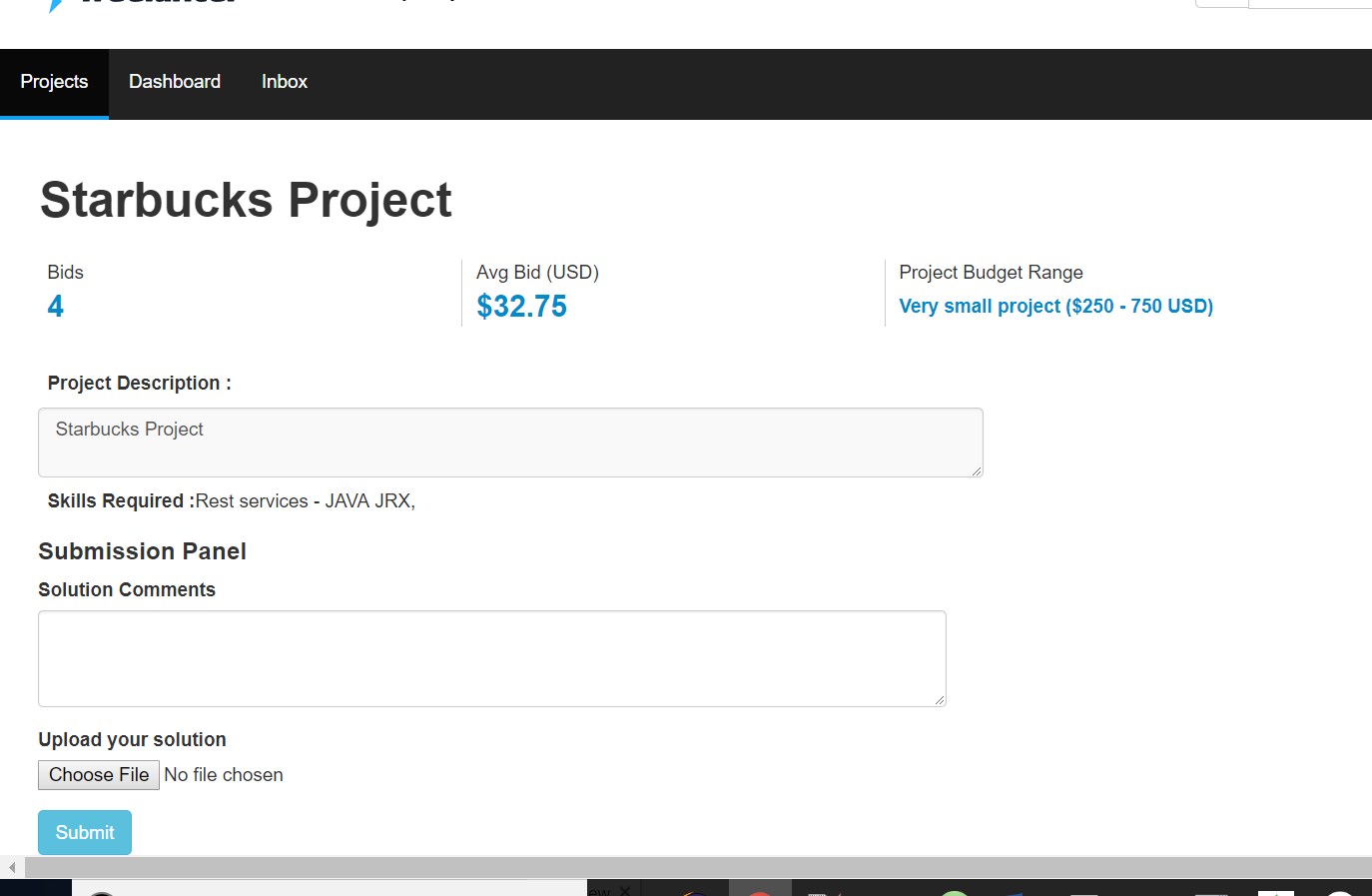
1. **After click on Hire Now button, Make Payment screen is visible to Employer.**

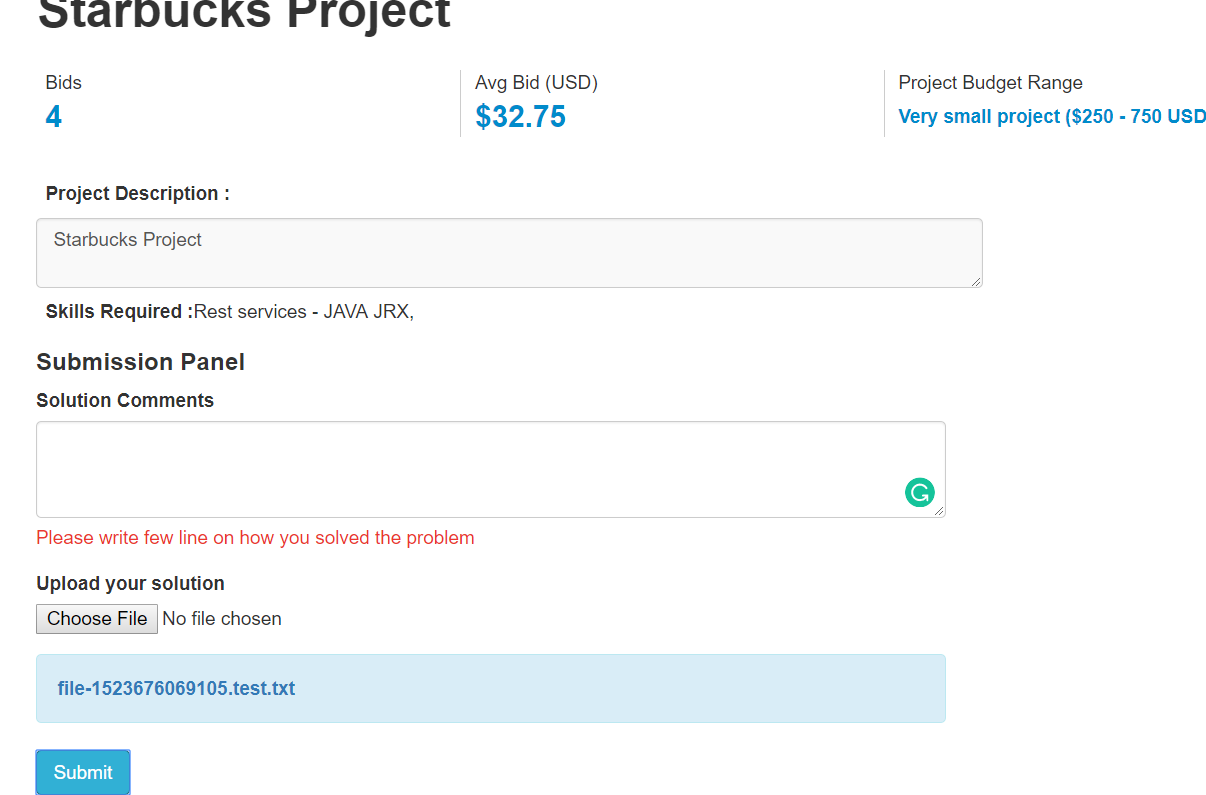


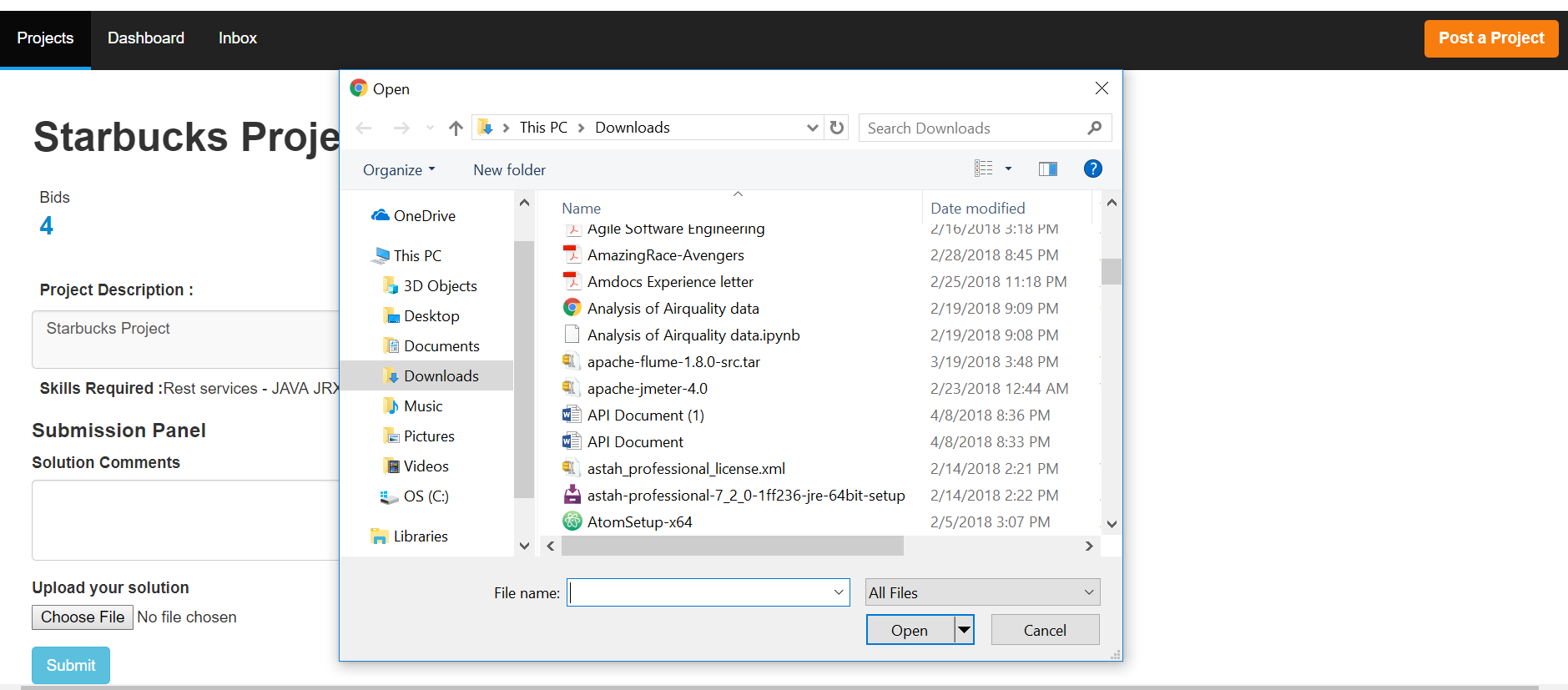
1. **If a freelancer is not hired, then for then this screen is visible**

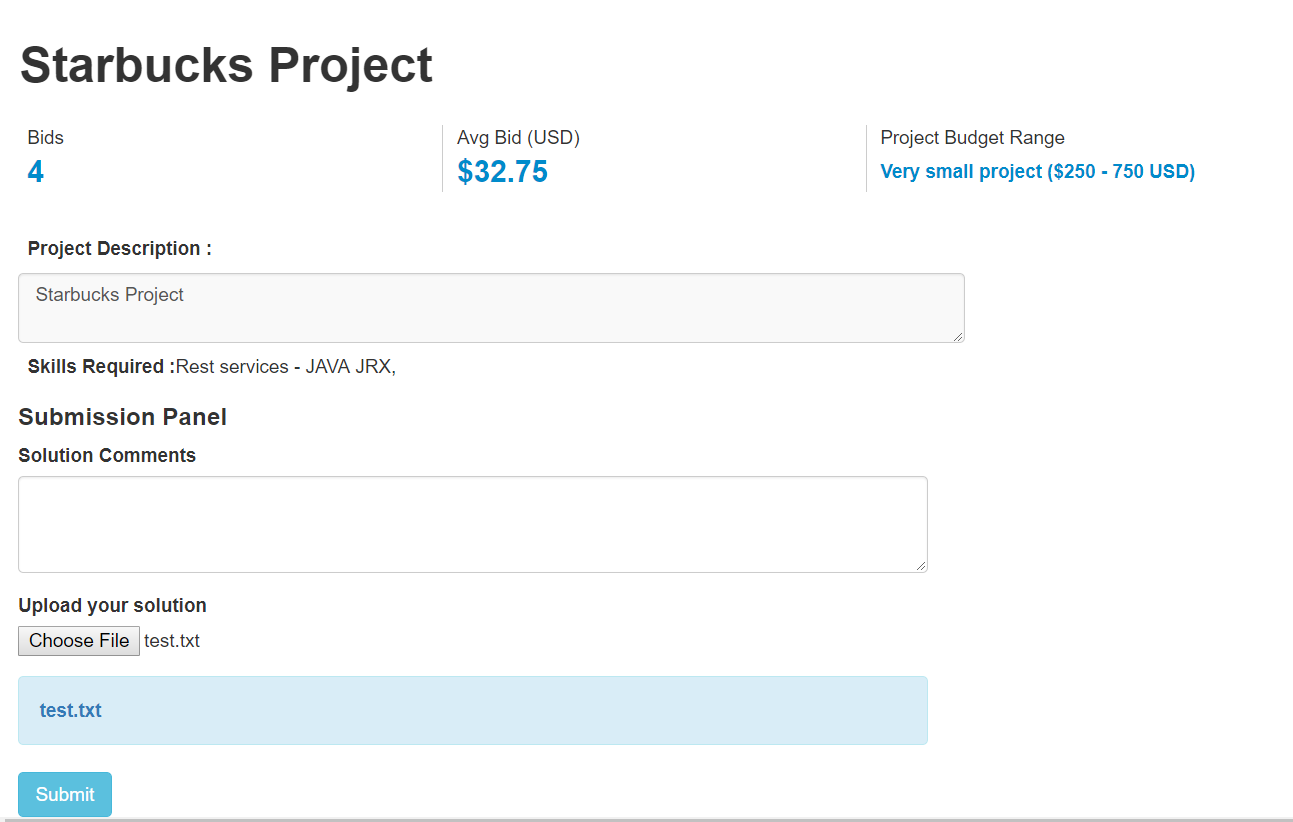


1. **If a freelancer is hired, the for them submission screen is visible (Validations included )**

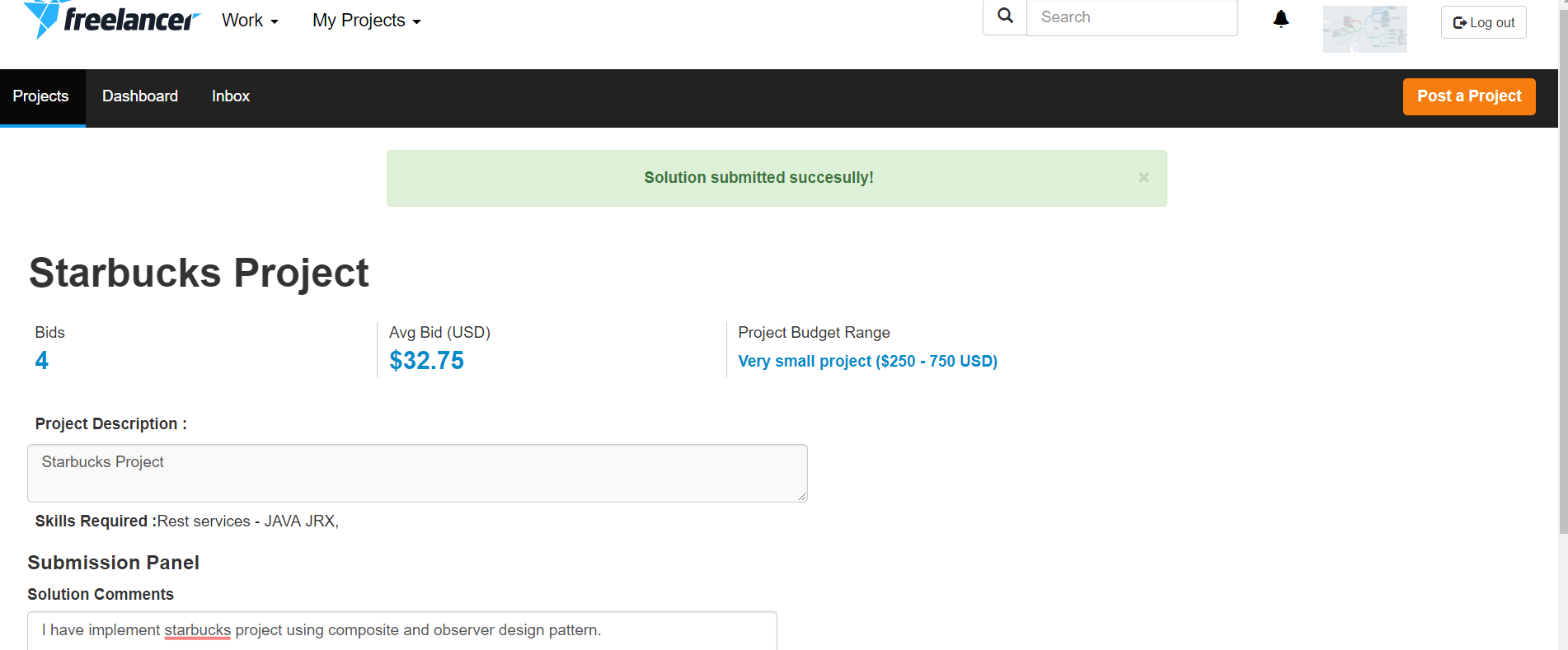




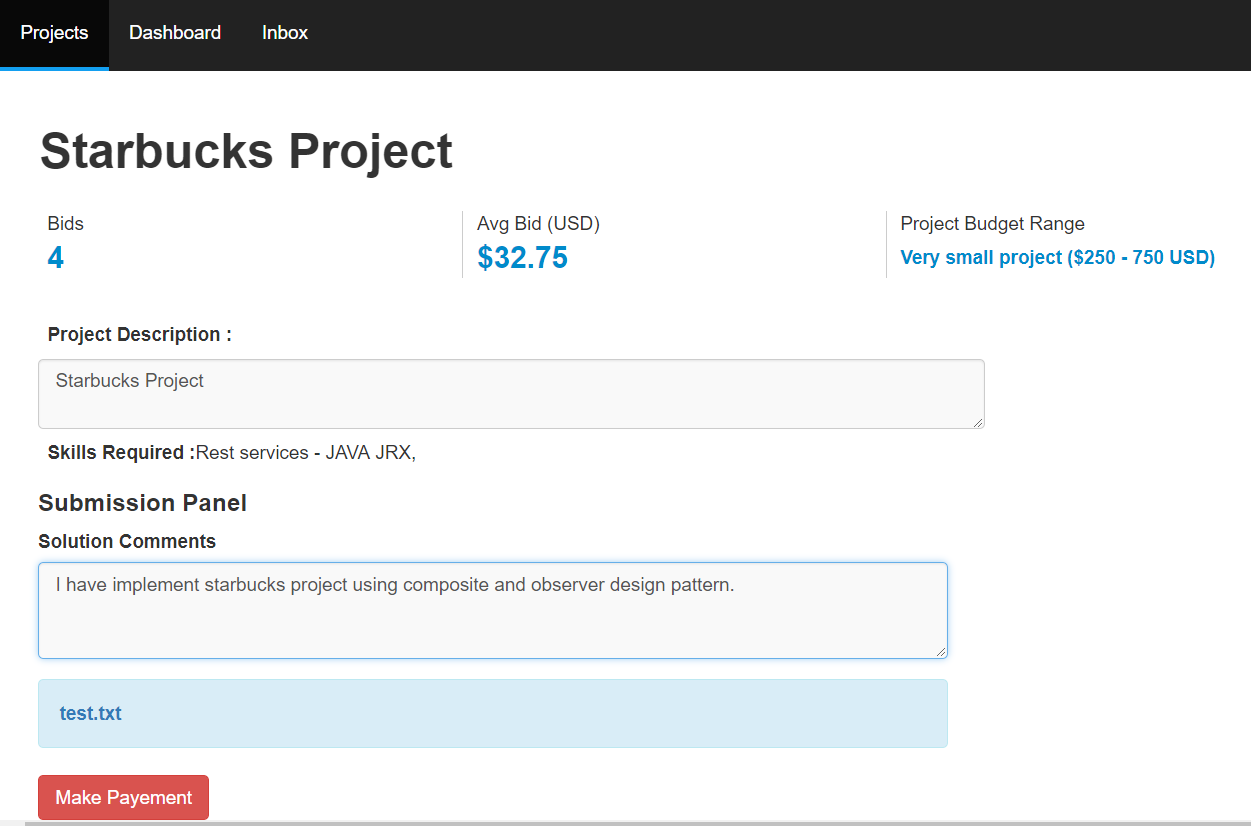


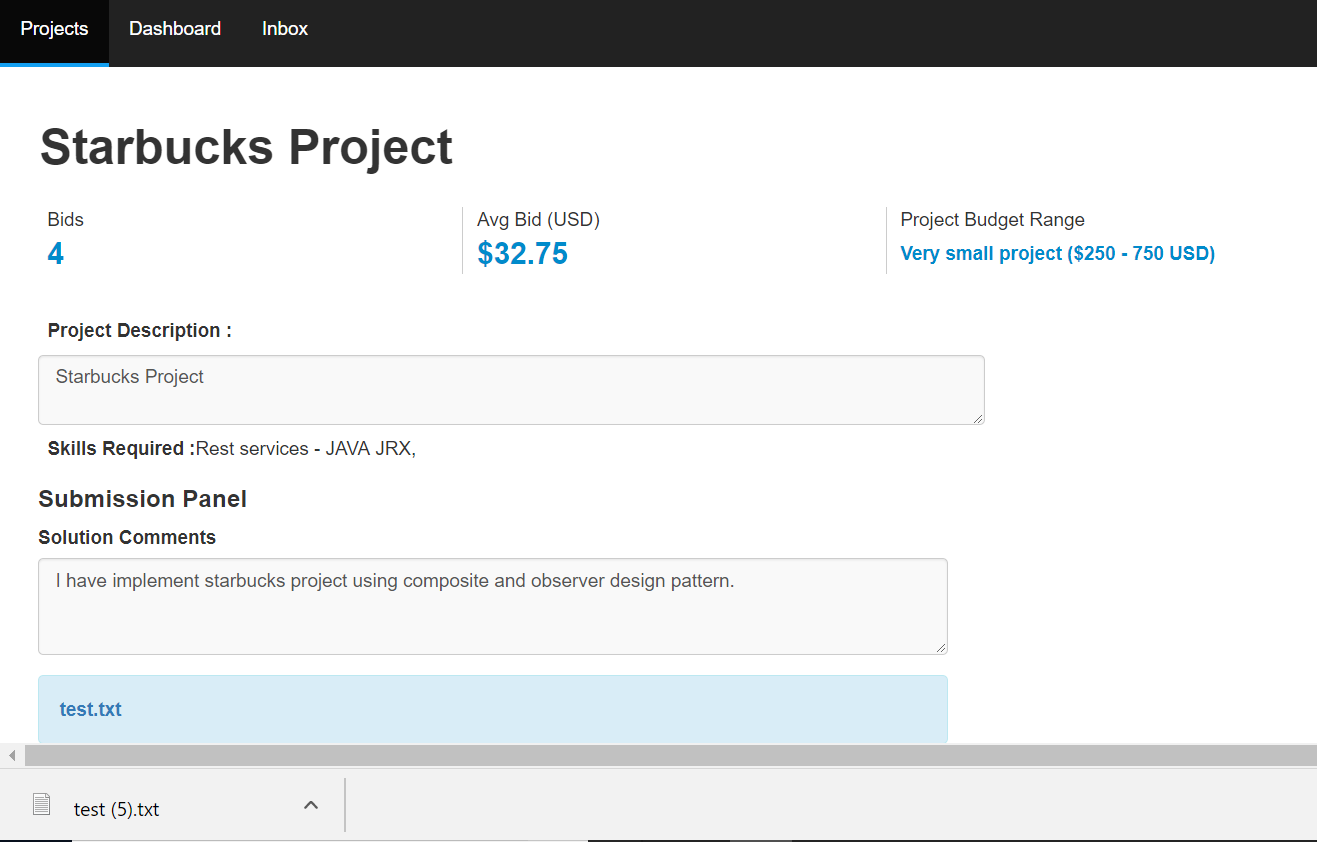


1. **Submit solution to Employer**

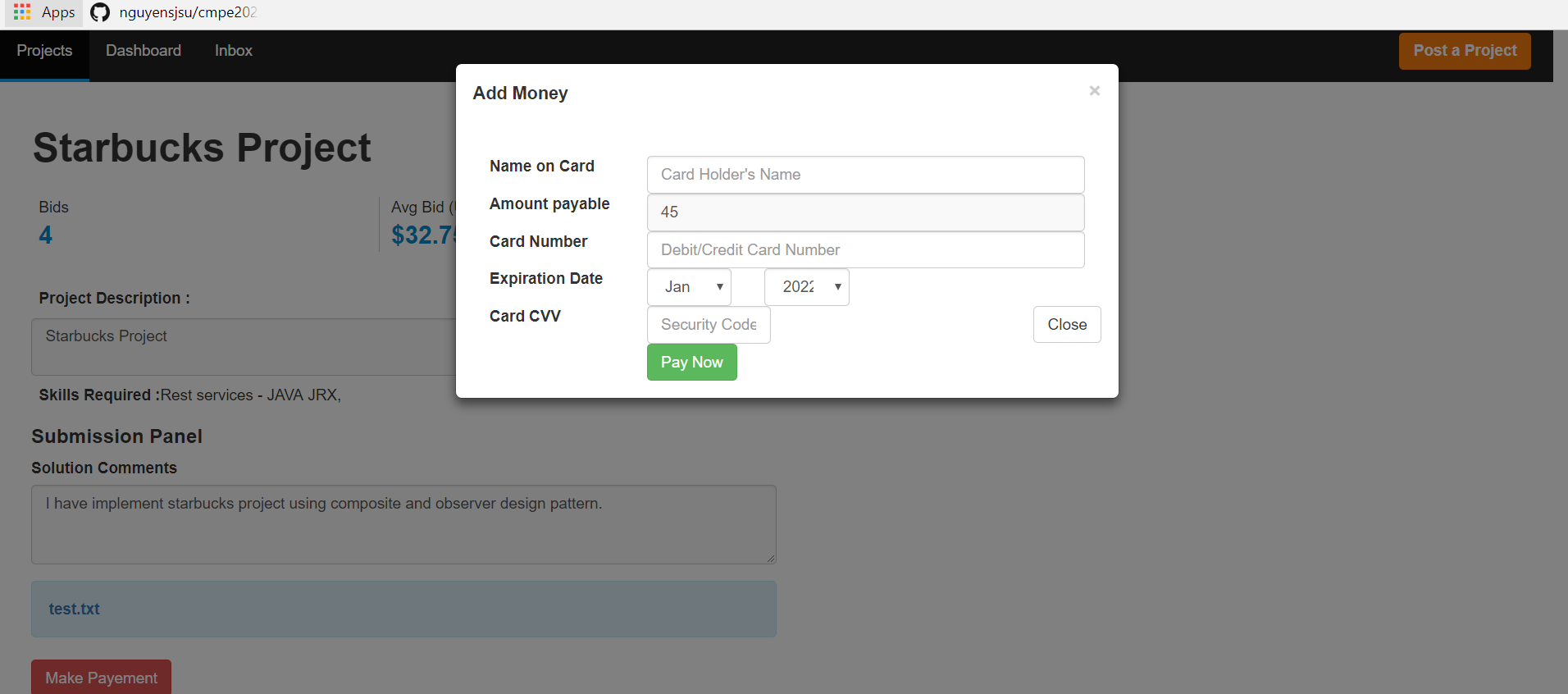


1. **Project Details screen for Employer after solution is submitted ( Files downloadable )**

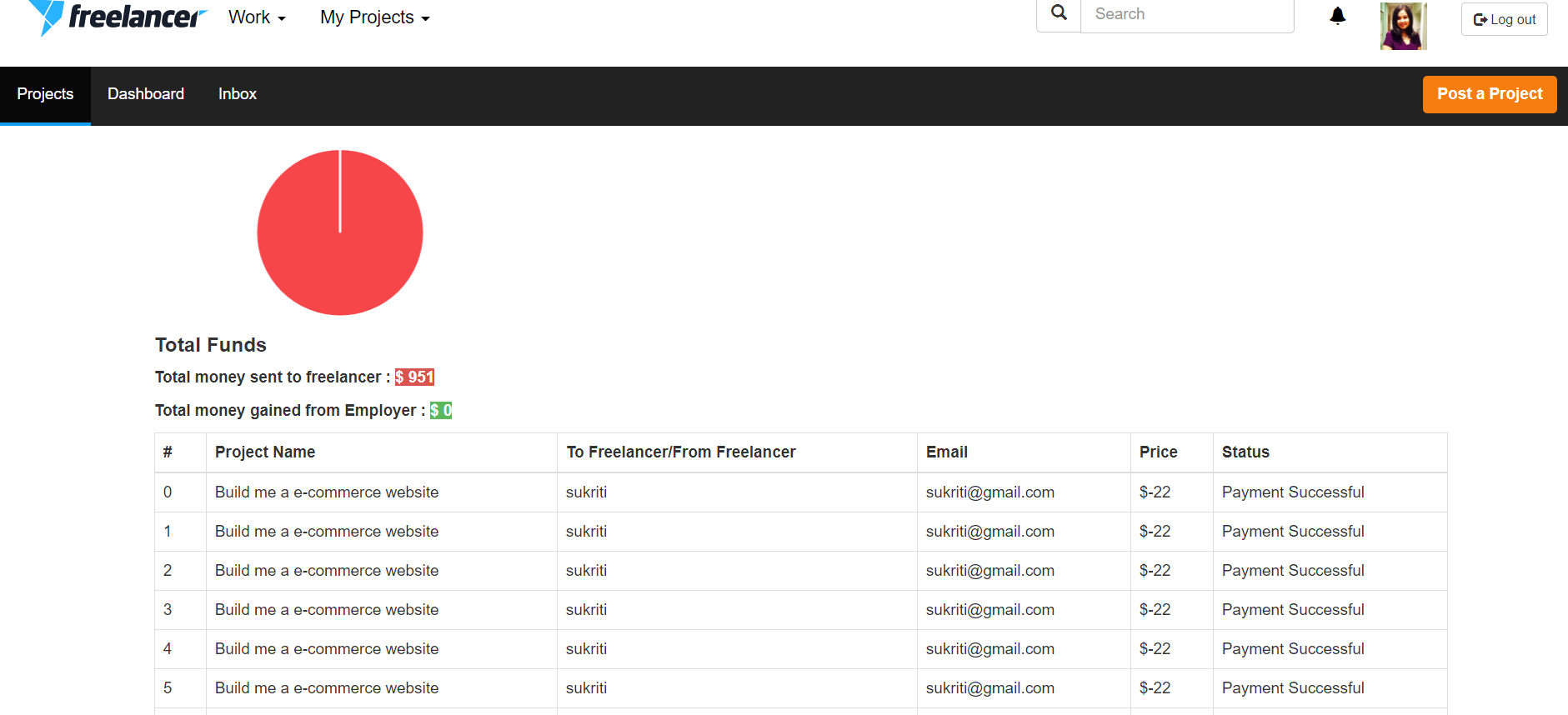




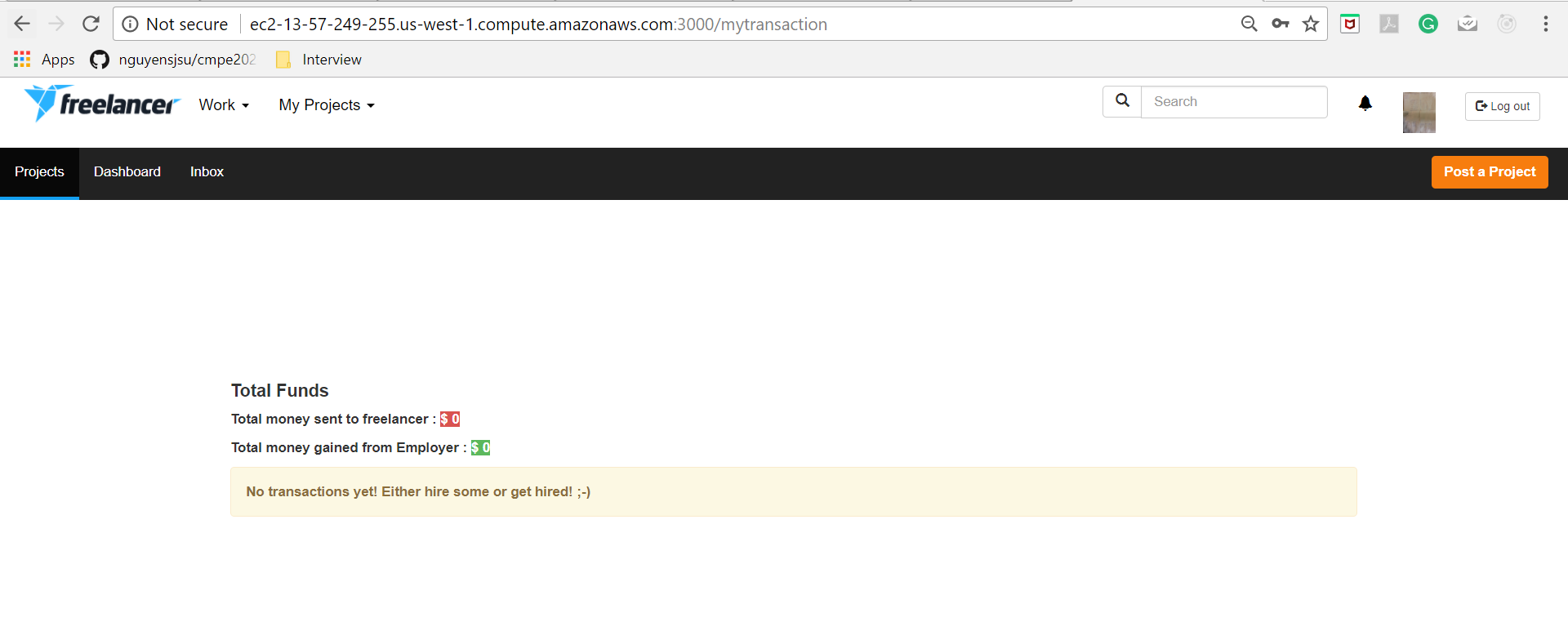
1. **Make Payment Modal with amount payable in disabled state ( same as bid price)**



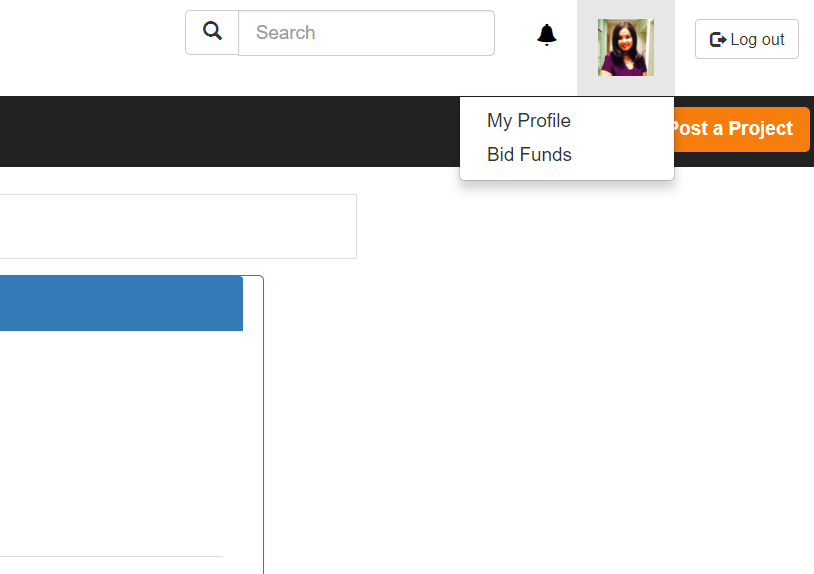
1. **Click on Pay Now button navigates to Transaction history page**



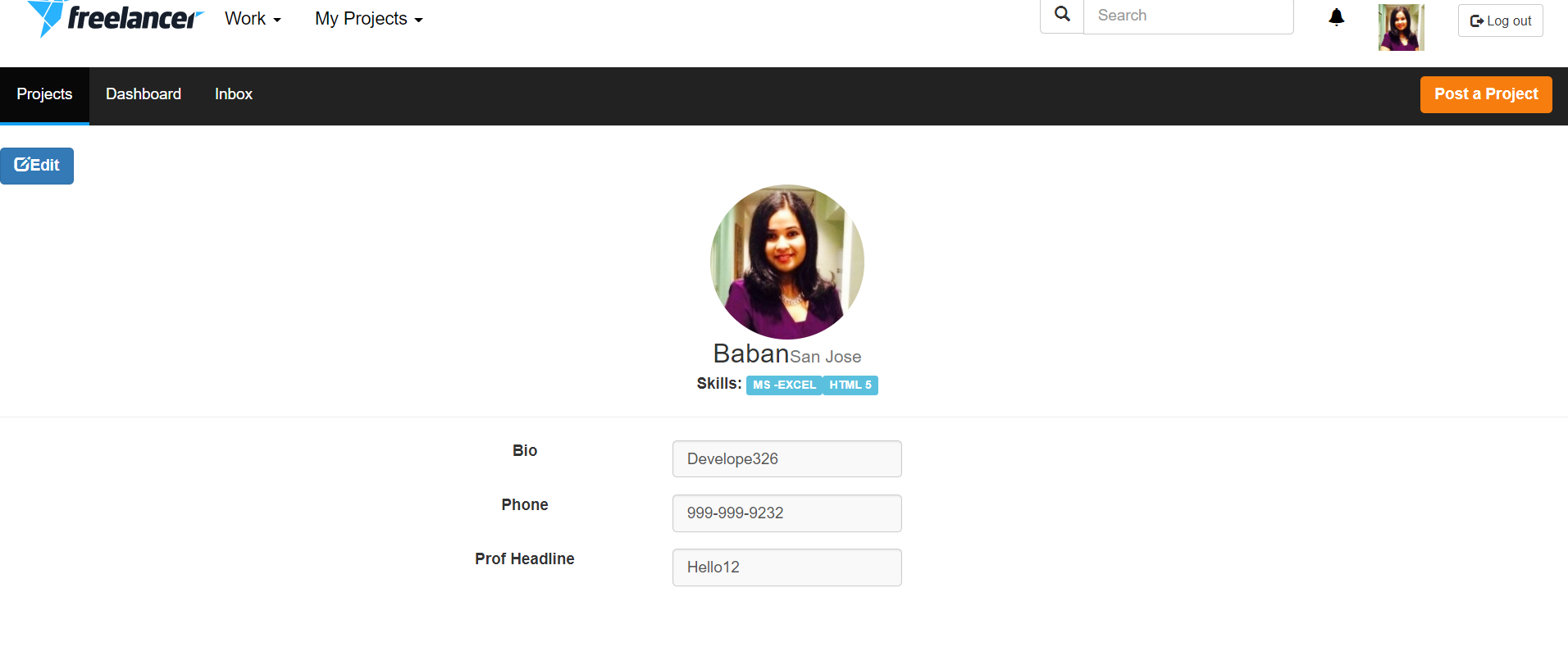
1. **Transaction history page when No transactions have been made**



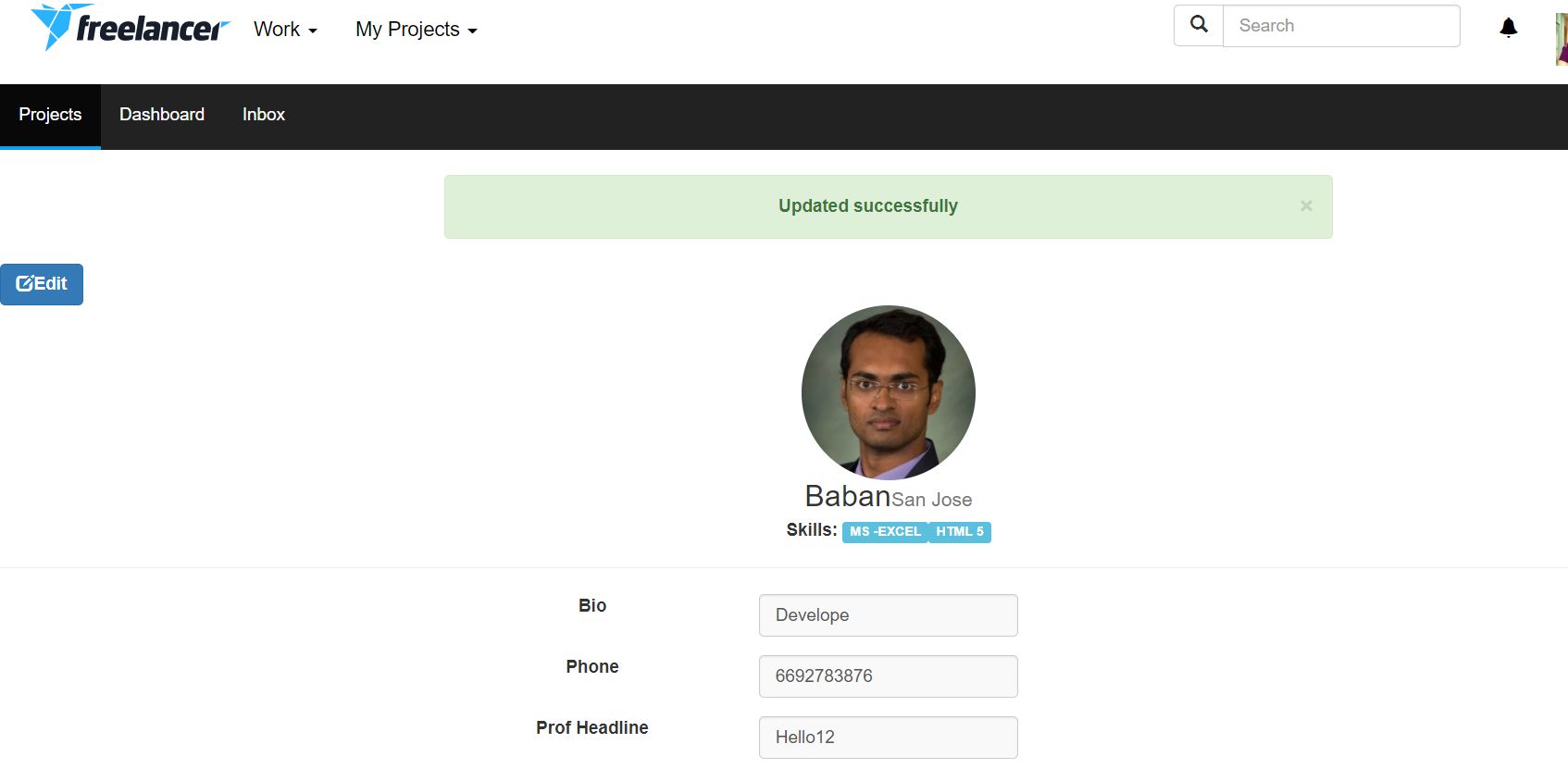
1. **User has option to see this funds from navbar menu as well.**



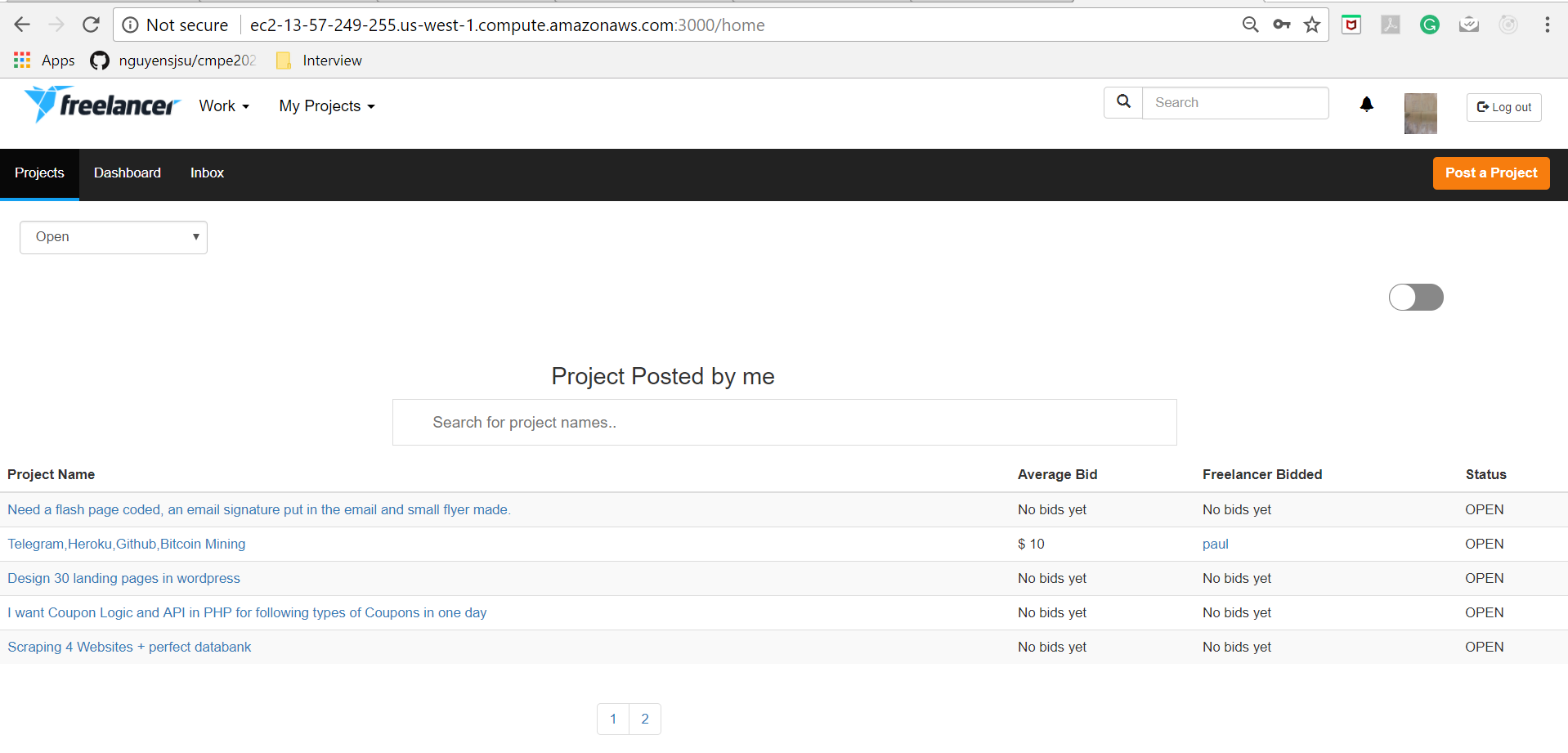
1. **View Profile**



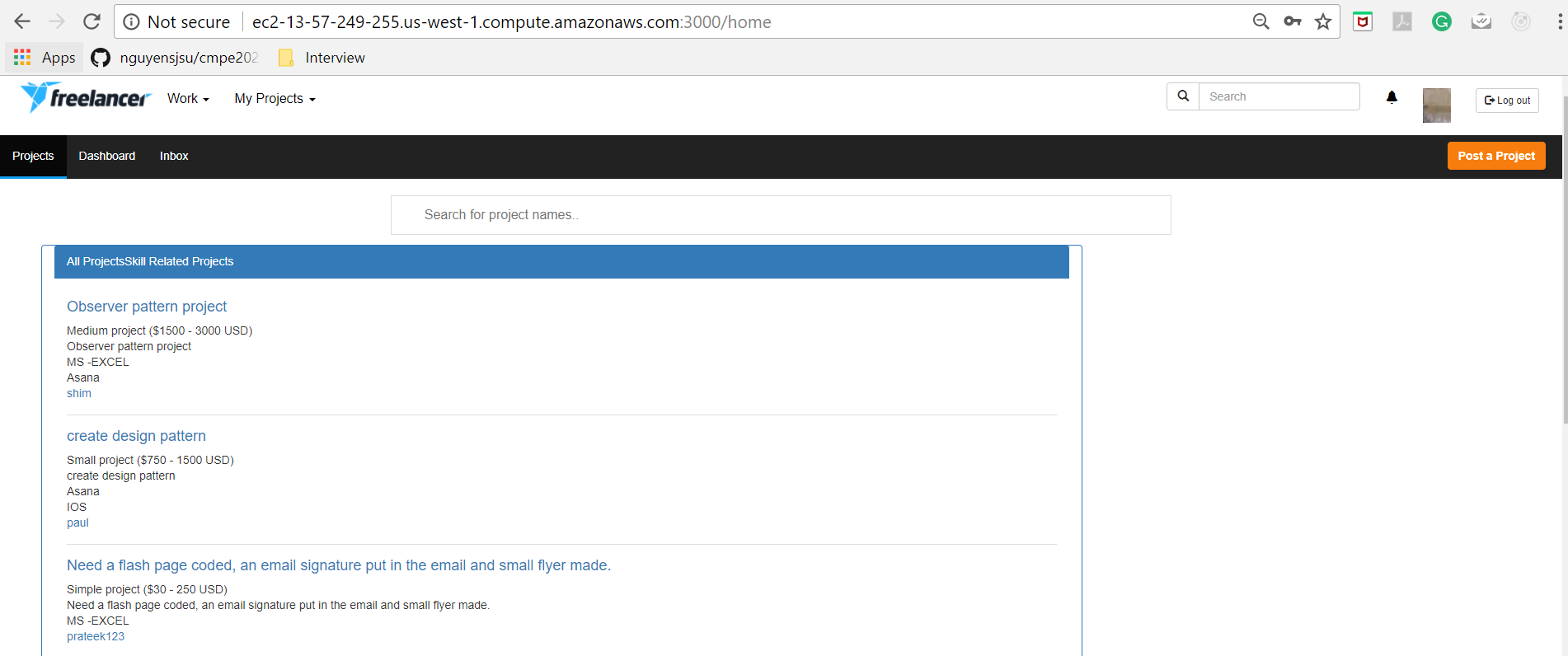
1. **Update Profile**

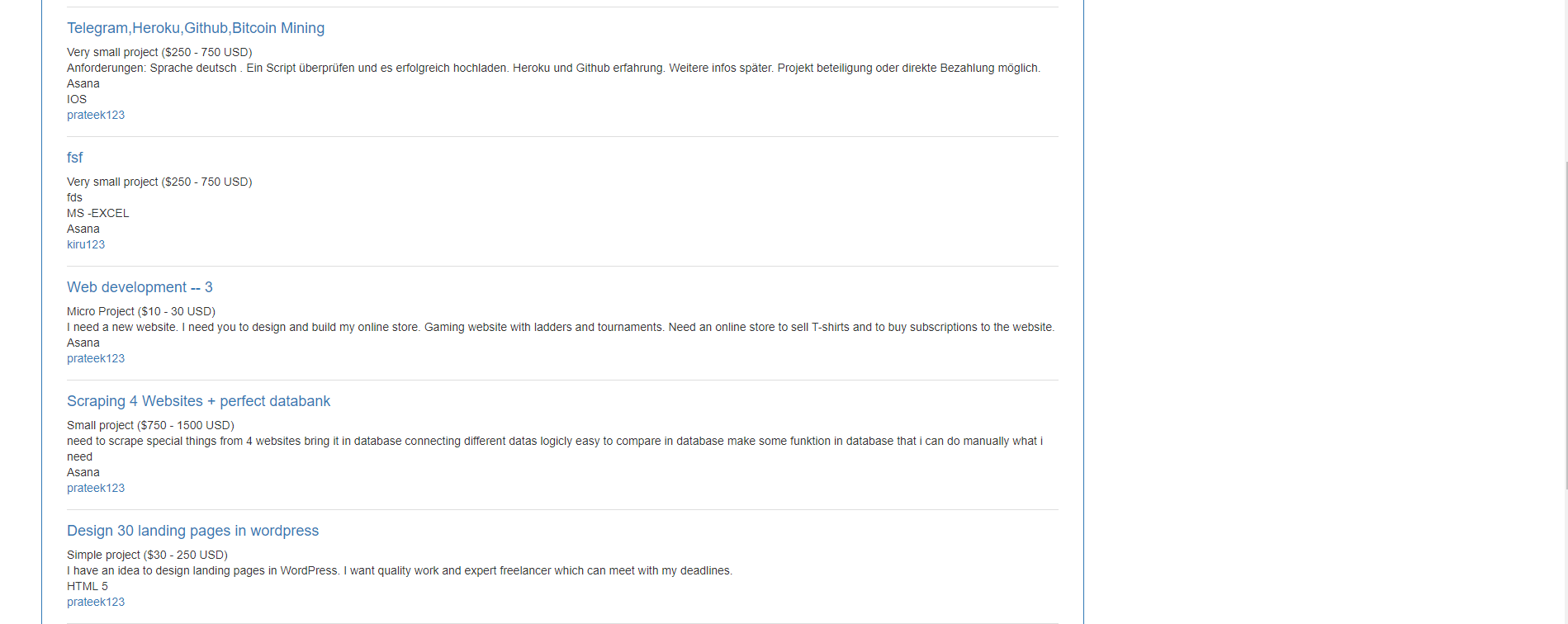


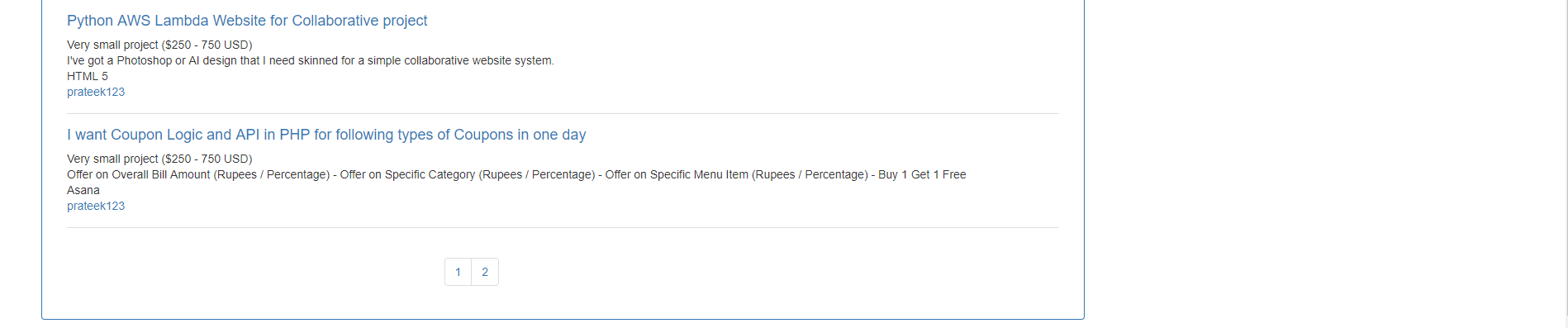
1. **Pagination on Projects posted by me on dashboard (5 projects per page )**



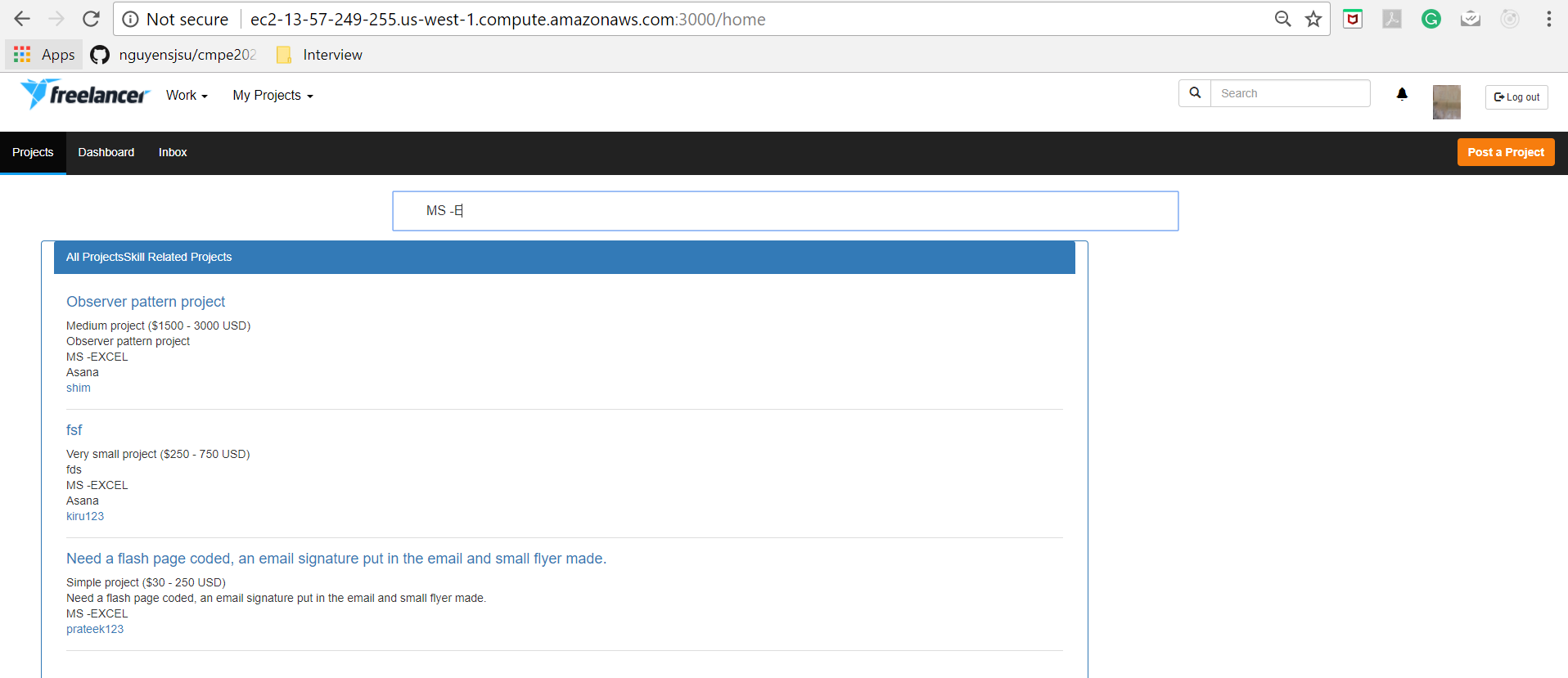
1. **All Open projects on Home page with pagination ( 10 projects per page )**

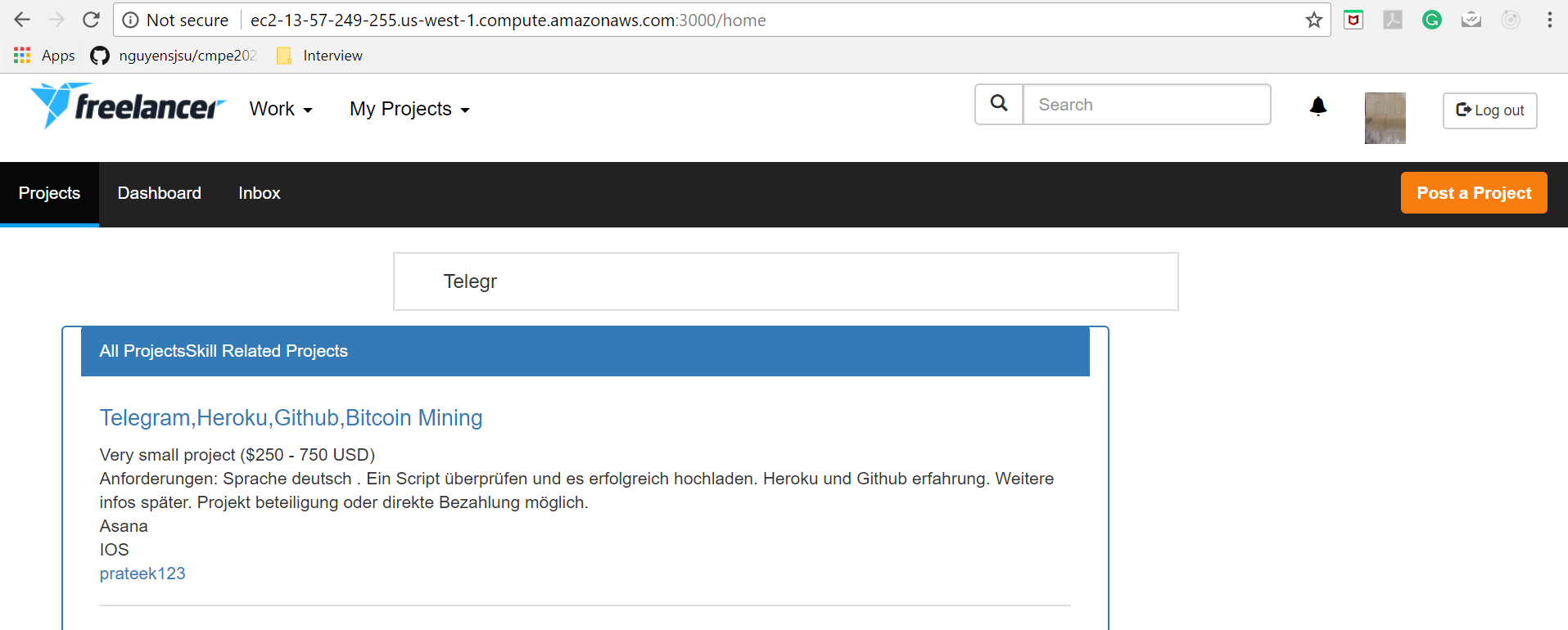






1. **All Open projects on Home page with search on project name and skill name**



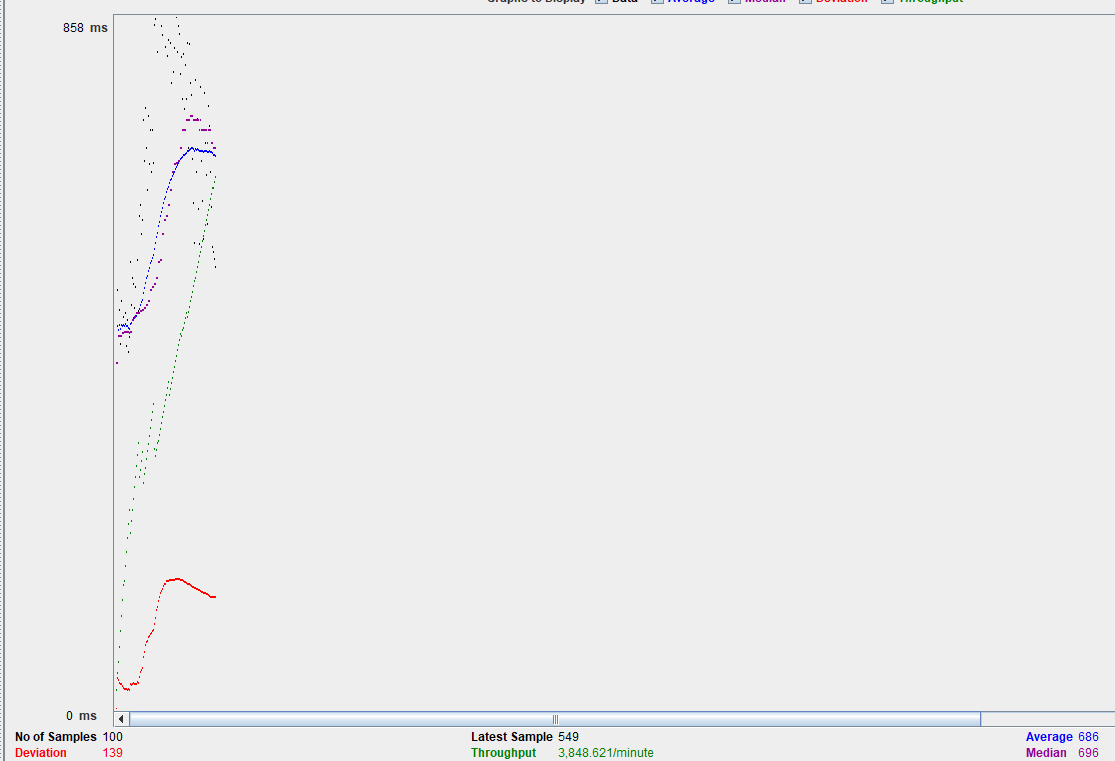


**Performance**

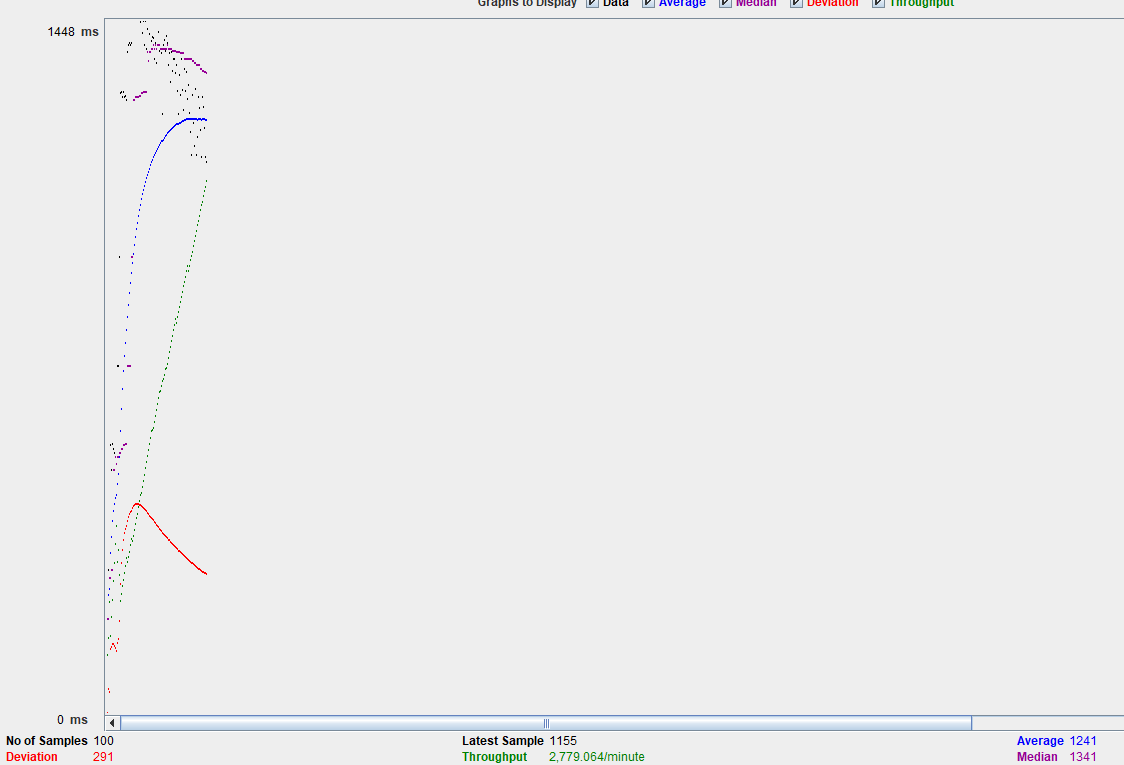
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **/skill/allSkills REST average time** | **100 requests** | **200 requests** | **300 requests** | **400 requests** | **500 requests** |
| **Without Connection Pooling** | 1241ms | 1647ms | 1431ms | 1615ms | 2314ms |
| **With Connection Pooling** | 686ms | 842ms | 917ms | 1090ms | 1406ms |

**Jmeter Testing**

1. **100 concurrent users skill/allSkills - with connection pooling (Average 686)**



1. **100 concurrent users skill/allSkills - without connection pooling (Average 1241)**



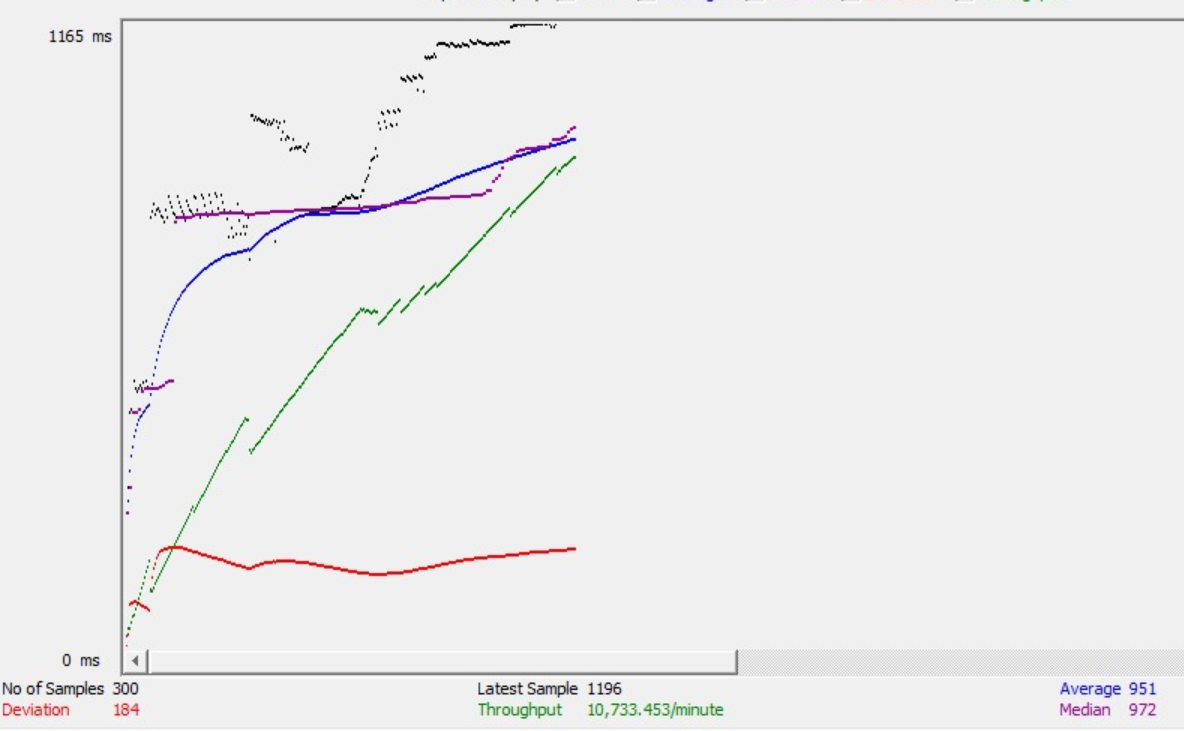
1. **200 concurrent users skill/allSkills - with connection pooling (Average 842)**



1. **200 concurrent users skill/allSkills - without connection pooling (Average 1647)**



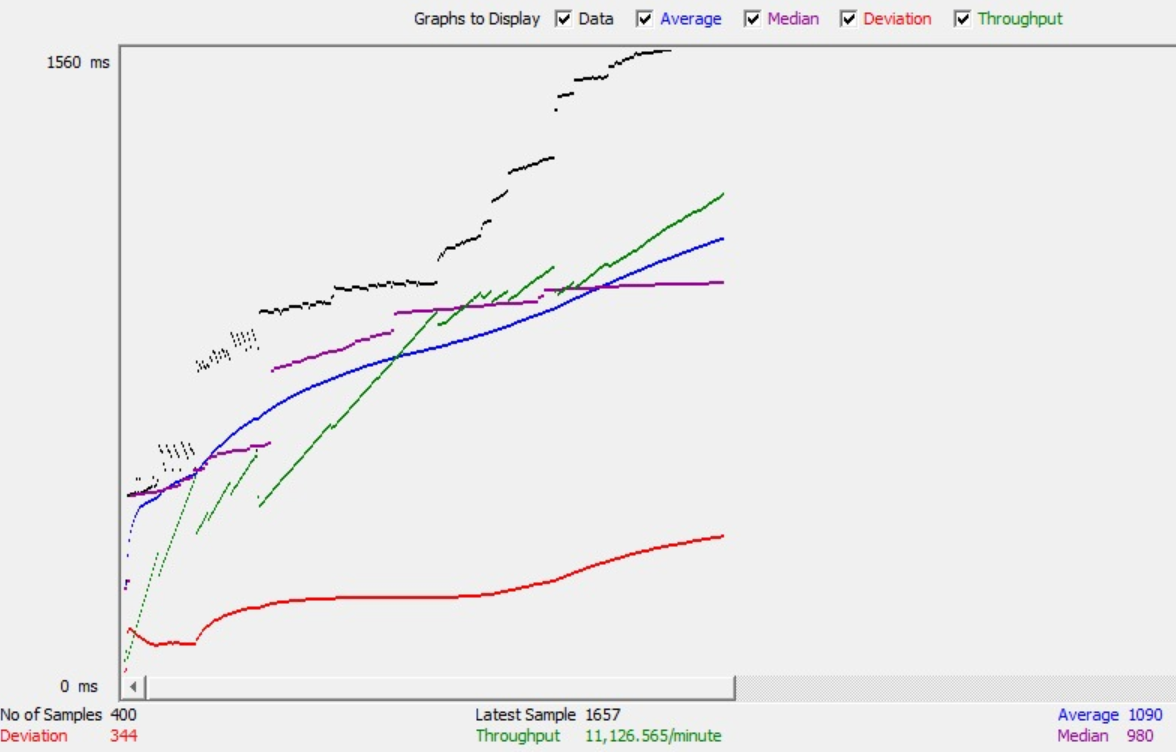
1. **300 concurrent users skill/allSkills - with connection pooling (Average 951)**



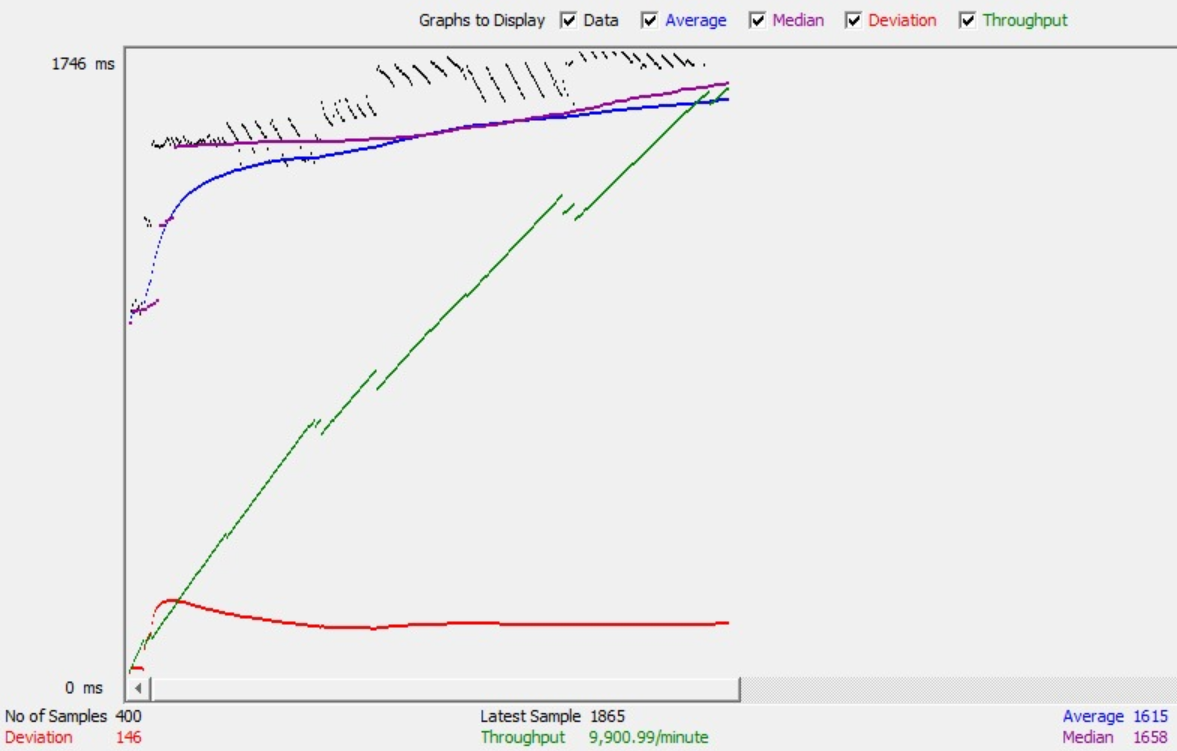
1. **300 concurrent users skill/allSkills - without connection pooling (Average 1431)**



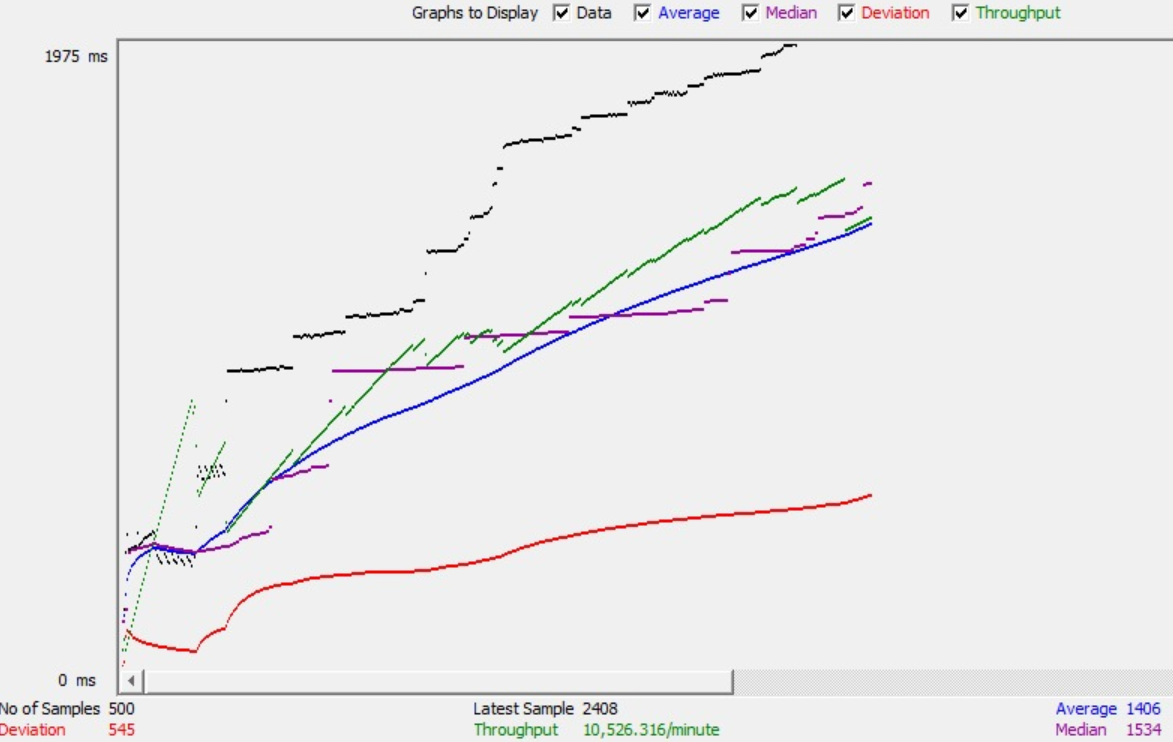
1. **400 concurrent users skill/allSkills - with connection pooling (Average 1090)**



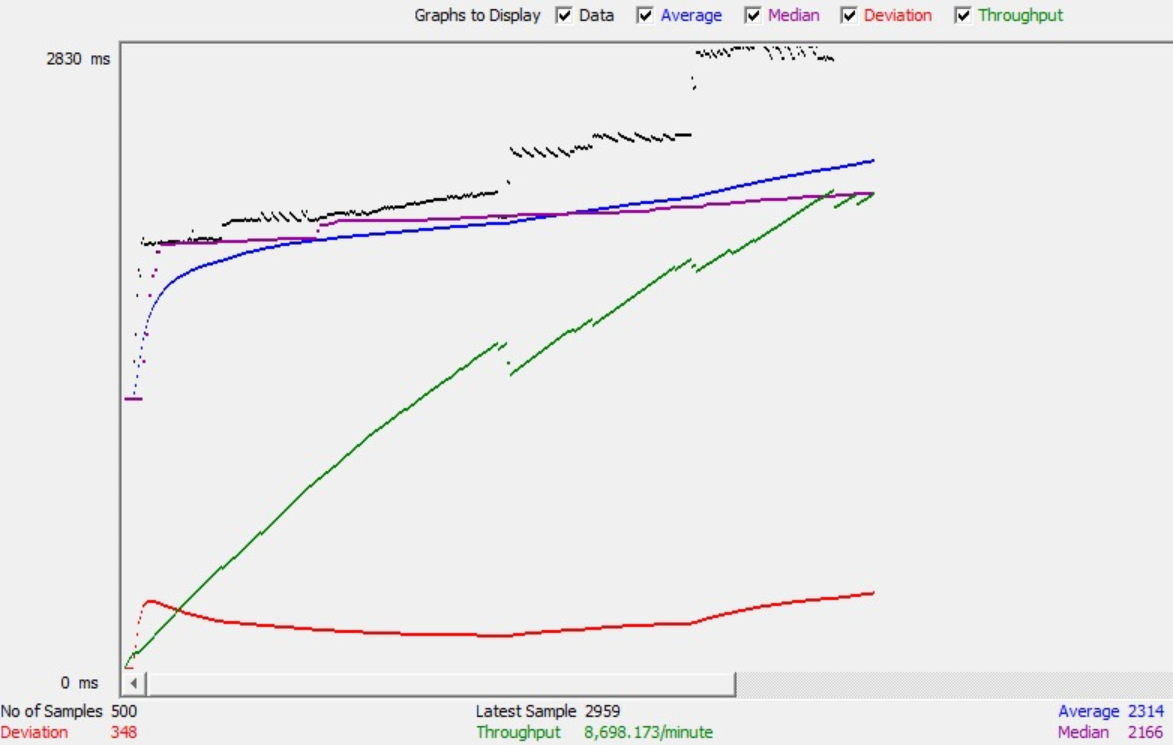
1. **400 concurrent users skill/allSkills - without connection pooling (Average 1615)**



1. **500 concurrent users skill/allSkills - with connection pooling (Average 1406)**

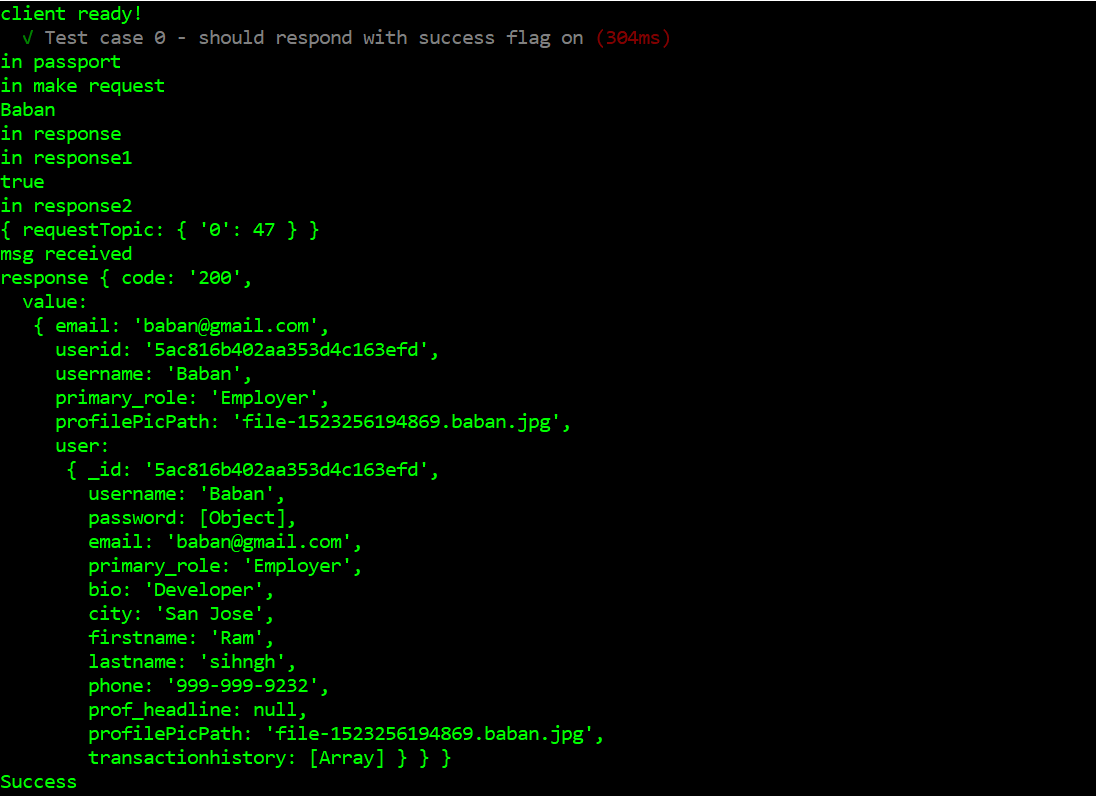


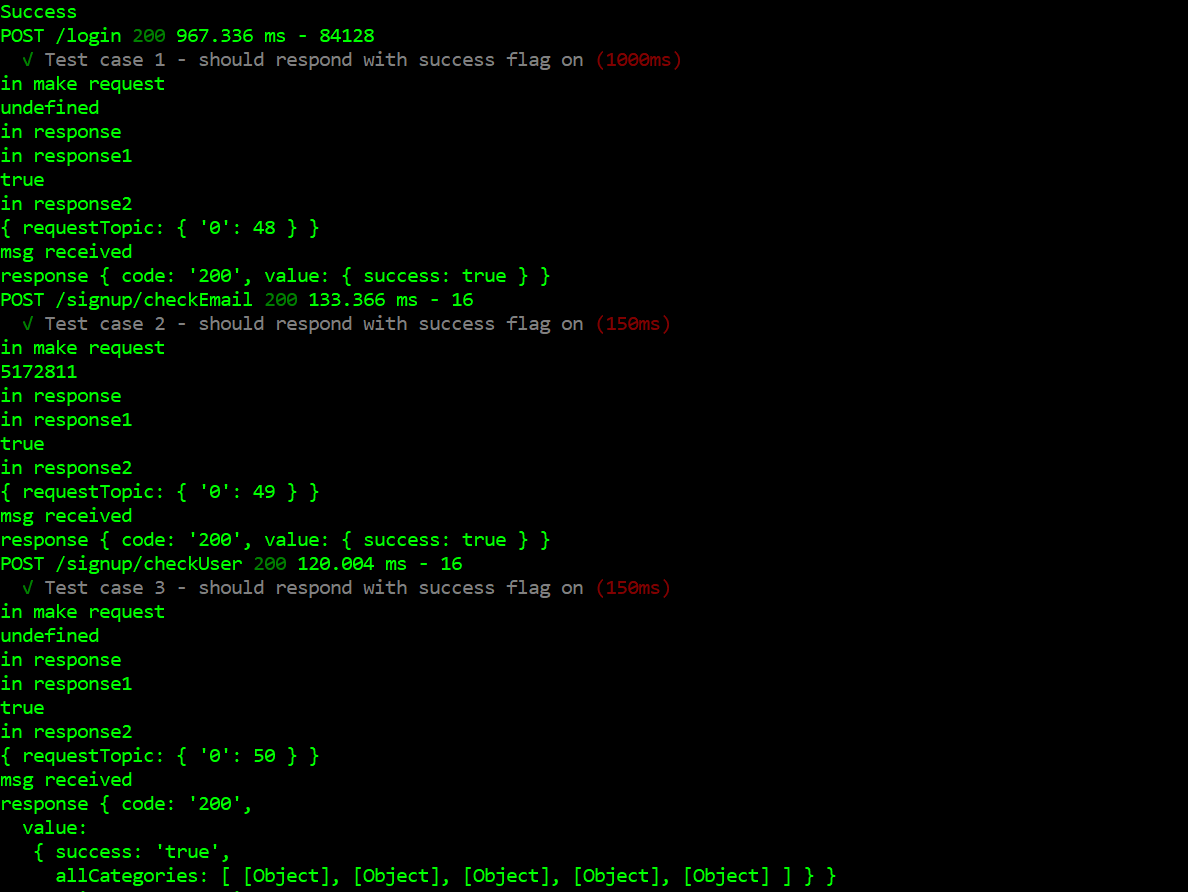
1. **500 concurrent users skill/allSkills - without connection pooling (Average 2314)**

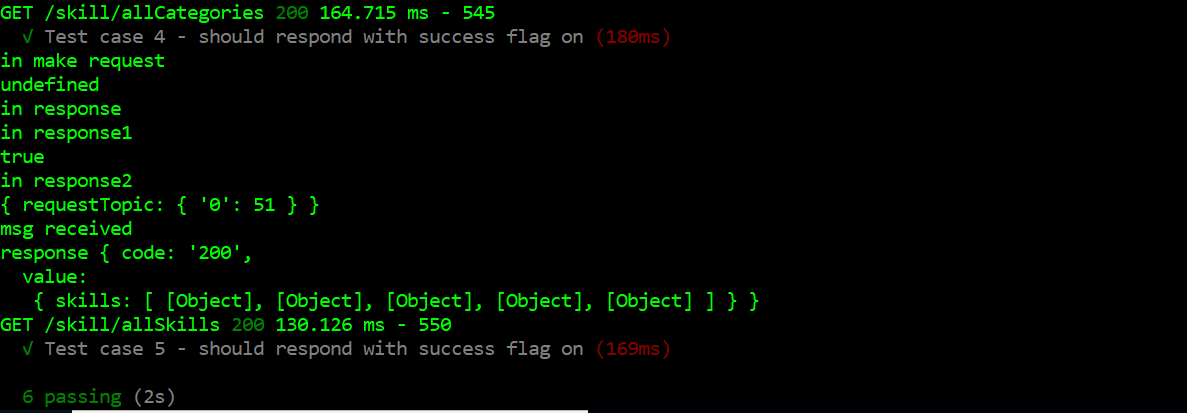


**Mocha Testing**

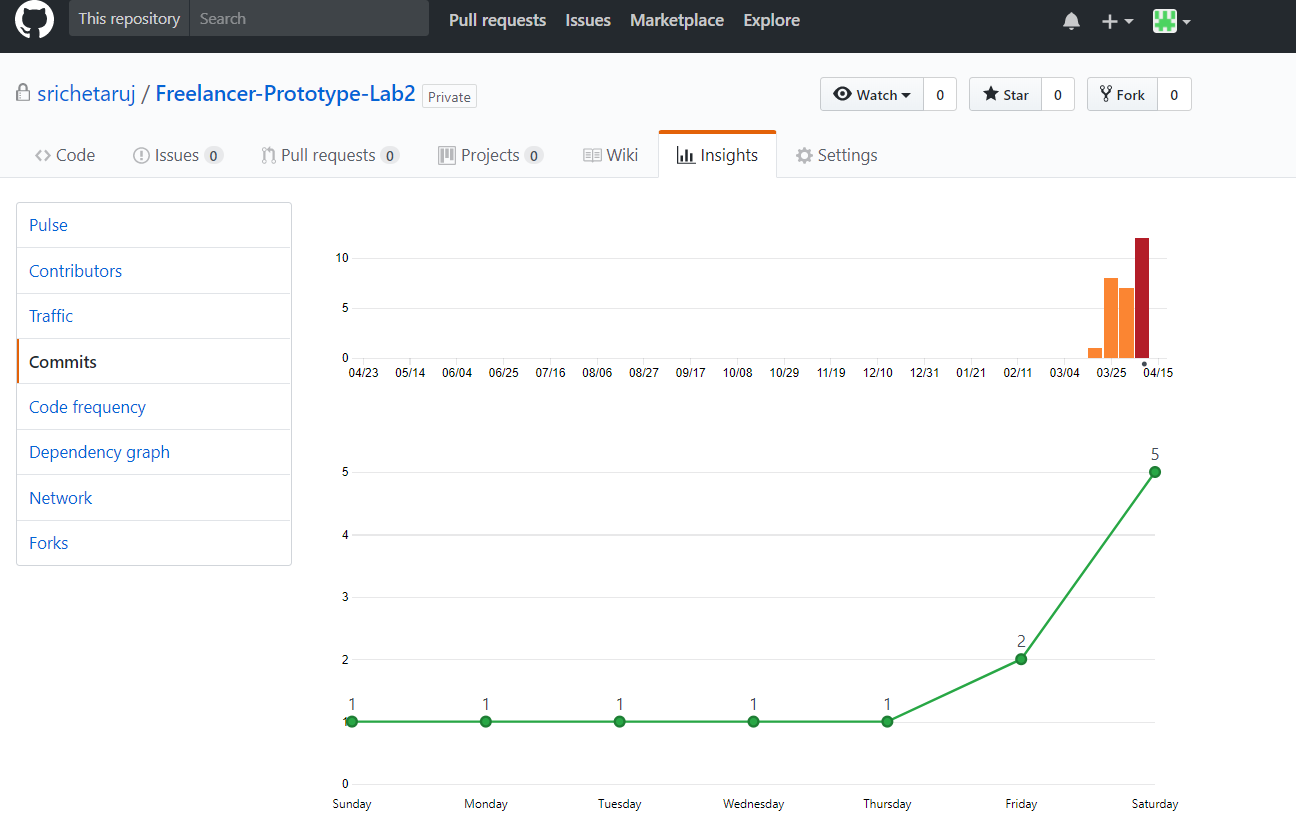
I have tested 6 Rest API in mocha







**Git commit history**



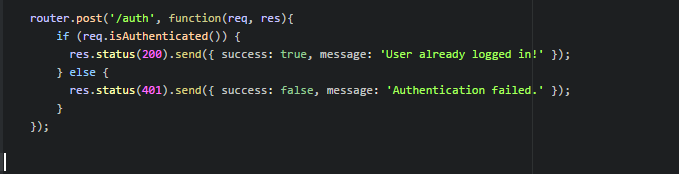
**Questions and Answers**

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

**Answer.**

I used JWT in Lab1 and passport in Lab2 for authentication. There is quite a lot of difference in their respective authentication processes.

* In Jwt we get the session token and store it into local storage and sent it to server with each REST api call(for validation). In Passport, it maintains the session data in database with expire time. So we need to call this below request every time to check whether request is authenticated or not.



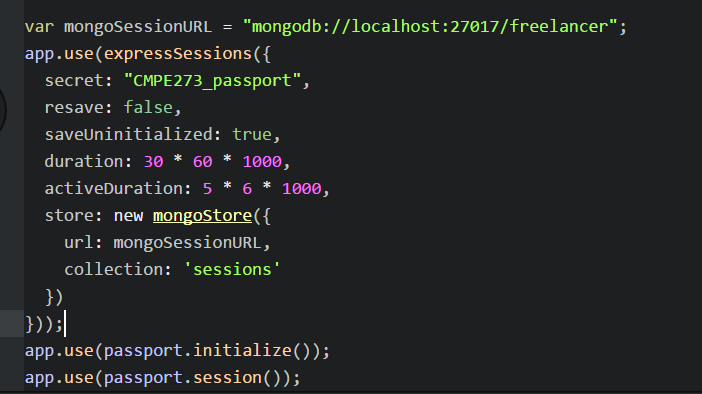
* Passport supports comprehensive set of strategies using a username and password, Facebook, Twitter, and more where as in JWT we don’t have such provisions. I have used local strategy.
* Passport also supports password encryption in built whereas while using JWT I used bcrypt in Lab1

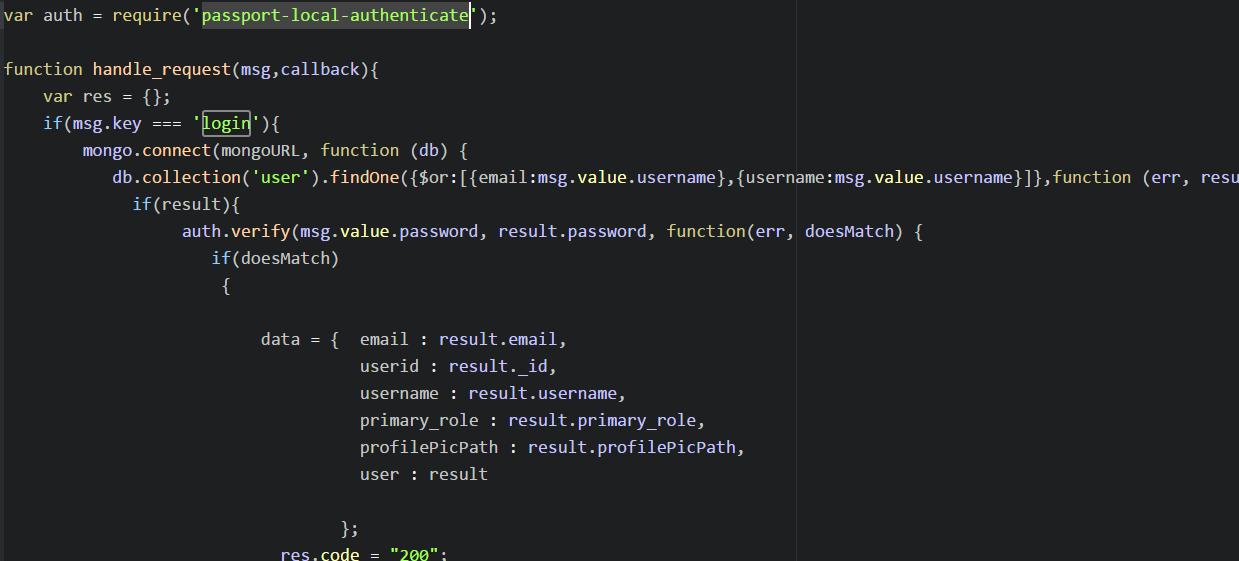
Passport is an authentication middleware for Node.js. It is extremely flexible and modular and can be dropped in to any Express-based web application using comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, and more. Using passport, the password in saved as object which has both salt and hash value saved in it. As Passport.js is built in connect, it supports session management as well like :

1. Serializing the authenticated user.

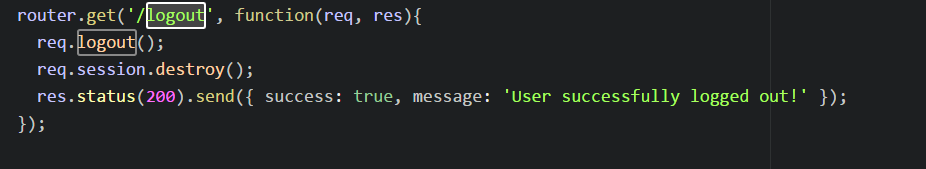


1. Managing the session





1. logging out



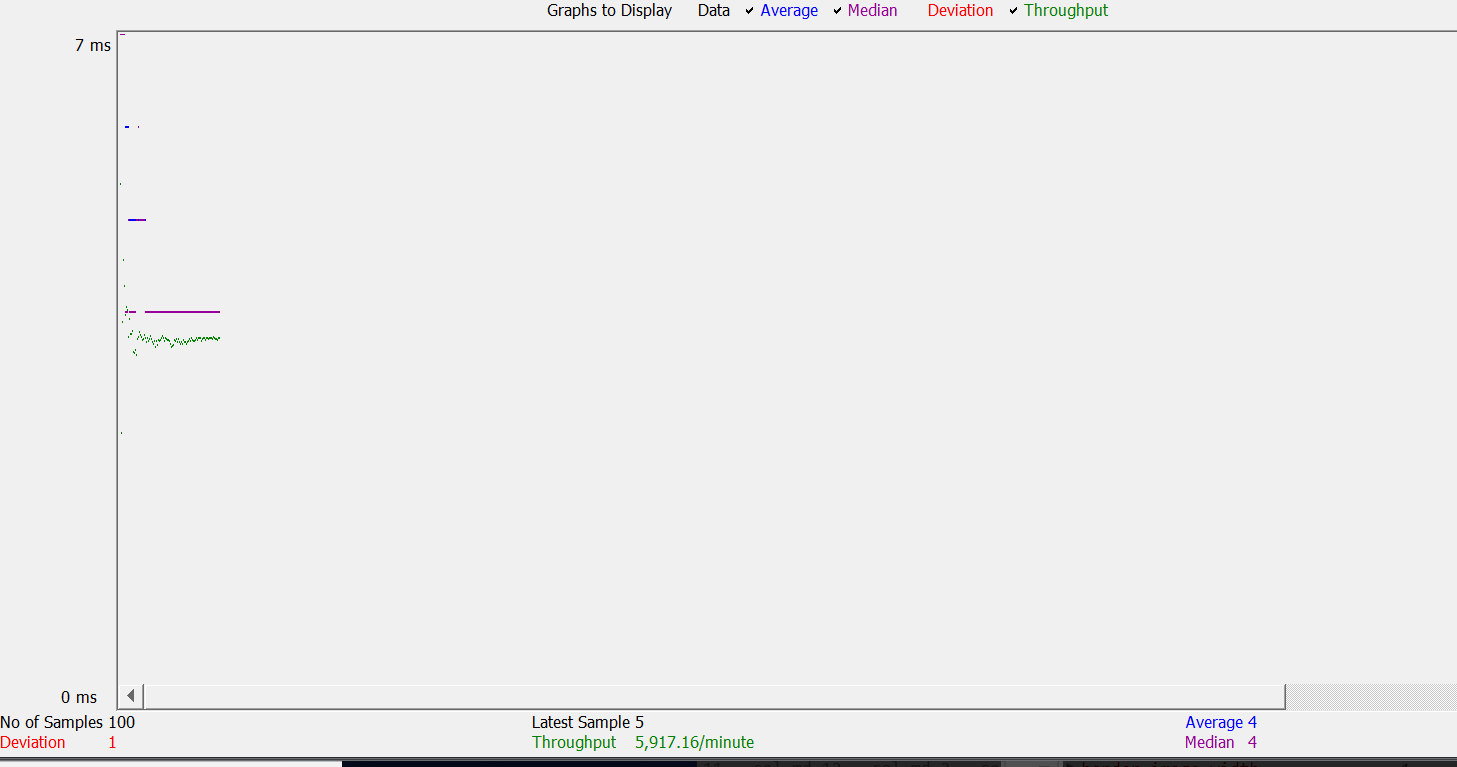
For Authentication I used the module - passport-local-authenticate.

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

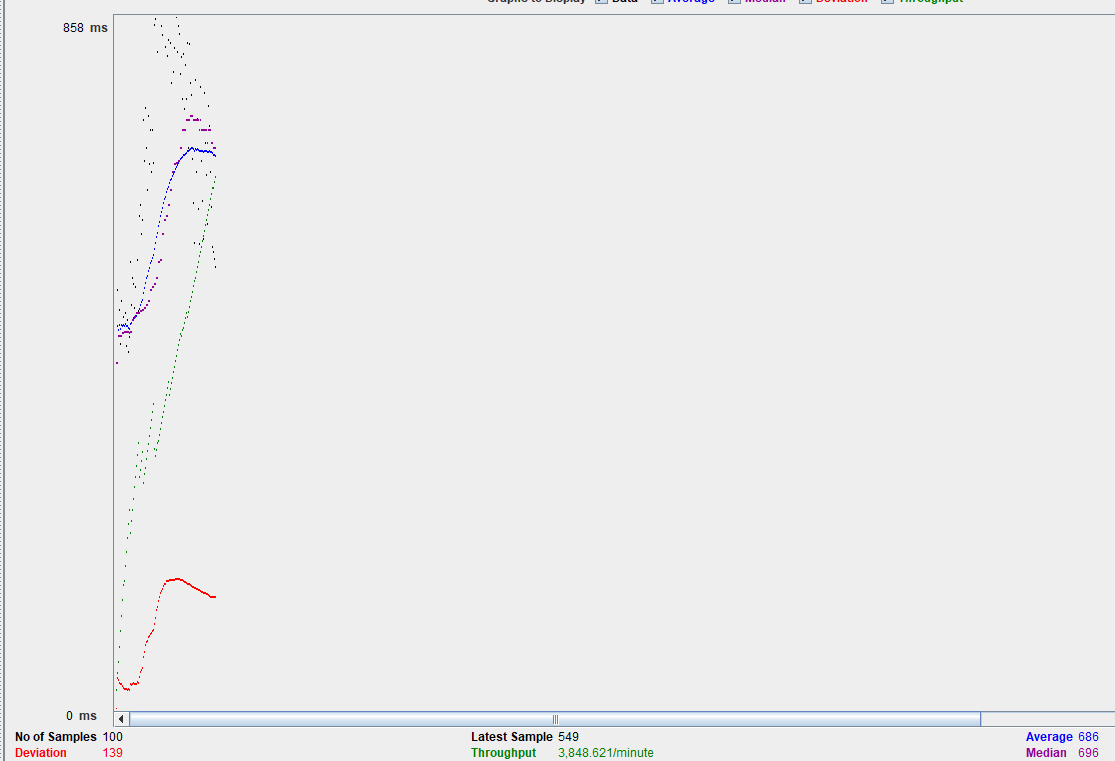
**Answer.**Kafka is a distributed streaming platform with three capabilities:

1. Publish and subscribe to streams of records, like a message queue or enterprise messaging system.
2. Store streams of records in a fault-tolerant durable way.
3. Process streams of records as they occur.

* **100 concurrent users skill/allSkills – without Kafka(Average 4)**



* **100 concurrent users skill/allSkills - with Kafka (Average 686)**



Above screenshots say that, with kafka running 100 /skill/allSKills requests takes more time than time taken without using kafka. Because after adding kafka between Node beckend server and mongoDB, Node backend server will add REST request as message to kafka broker and when it will be consumed at kafka backend server it will reach mongoDB. Same path is followed back when response is sent back from mongoDB to Node backend server. So, because of this one one more step, REST API takes more time.

However, Kafka is good with scalability and fault tolerance.

I created 2 topics to store the messages. i.e. requestTopic and responseTopic.

I created topic with replication factor to 3 to support load balancing and a fault tolerant system. Whenever the message was failed to get from the broker, 2 more replicas would be helpful to get the message back. Producers get an acknowledgement back when they publish a message containing the record's offset. The first record published to a partition is given the offset 0, the second record 1, and so on in an ever-increasing sequence. Consumers consume data from a position specified by an offset, and they save their position in a log by committing periodically.

Kafka is also persistent. Messages are immediately written to the filesystem when they are received. Messages are not deleted when they are read but retained. This increases performance.

 In lab1, I did not use Kafka, data from node server directly was passed to sql database and vice-versa. It was working fine but without load balancing. Every time I call sqldatabase, the data is never cached. I need to again fetch the same data even if it was delayed or anything. Nothing remains in queue.

**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.**

**Answer.** If given an option to implement both MySql and MongoDB, then I would put those data in sql whose atomicity (ACID properties) is important to preserve.

Mongo DB has following characteristics:

1.      Better performance.

2.      High speed and used for scalability

3.      Slow lookups (i.e. joins)

 Taking these advantages into consideration, I would put those records into mongoDB which don’t have dependency on any data. For example, fetching all the skills available category wise. i.e. I would put skill and skill\_category table in Mongo DB.

On Home screen, the requirement was to fetch those projects whose skills match with the skills of user. Also, user can update their skill anytime. So if a user would change their skill then the resultant project would change consecutively. But in that resultant json will still be the same. So atomicity is sacrificed.