**CMPE 273 – Lab 2**

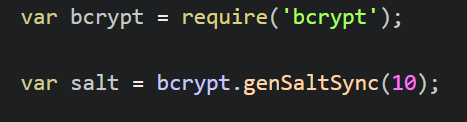
**Sricheta Ruj (SJSU ID - 012527606)**

**Questions :**

**1. Compare passport authentication process with the authentication**

**process used in Lab1.**

**Ans.** In Lab1, I used brcypt in authenticate the user. The password entered by the user is stored as hash value in the database. I have used bcrypt for encrypting password when a users sign’s up in the application. It hashes the password. Bcrypt is available in npm package manager. It is easy to hash and compare passwords. It uses the Eksblowfish algorithm to hash passwords. It is adaptive hash algorithm. Any state depends on both salt and key which is the password which I have used to generate the hash. Hence, no state can be precomputed without the knowledge of both. Without the knowledge of the salt and password, one cant get the credentials. The salt is computed by passing the rounds which is the cost of processing data.

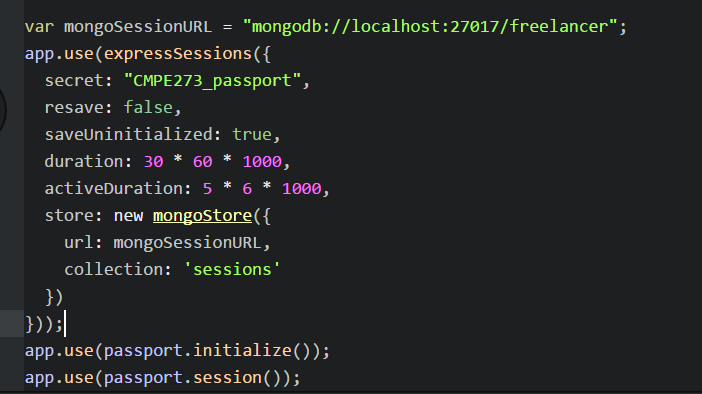


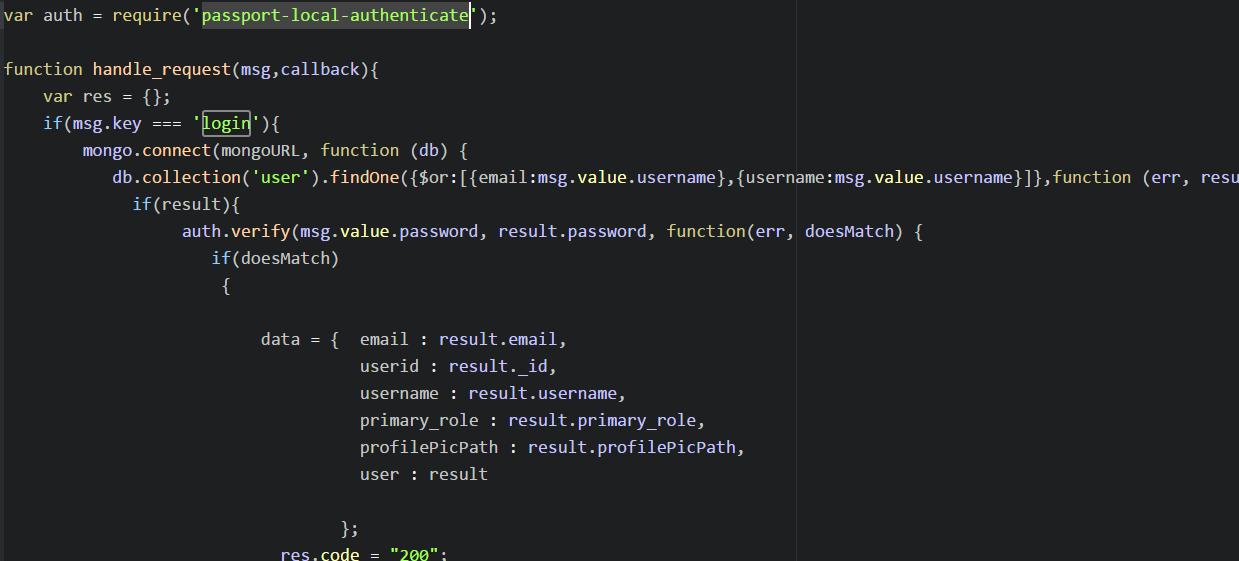
In lab2, I used passport.js for authentication. Passport is an authentication middleware for Node.js. It is extremely flexible and modular and can be dropped in to any Express-based web application using comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, 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.

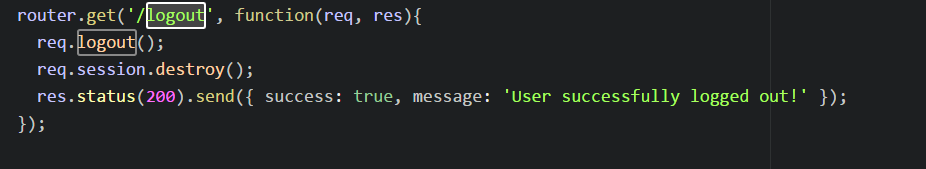


1. Managing the session





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.**

**Ans.**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.

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.

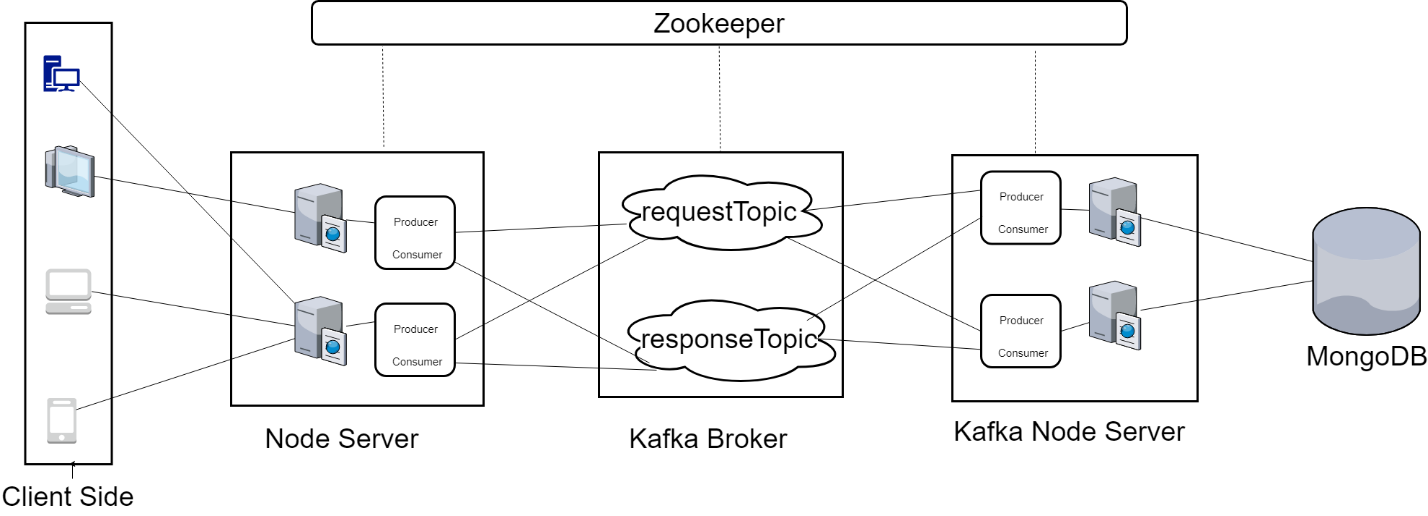
Kafka is also persistent. Messages are immediately written to the filesystem when they are received. Messages are not deleted when they are read but retained. This increases performance.

Without Kafka, 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**

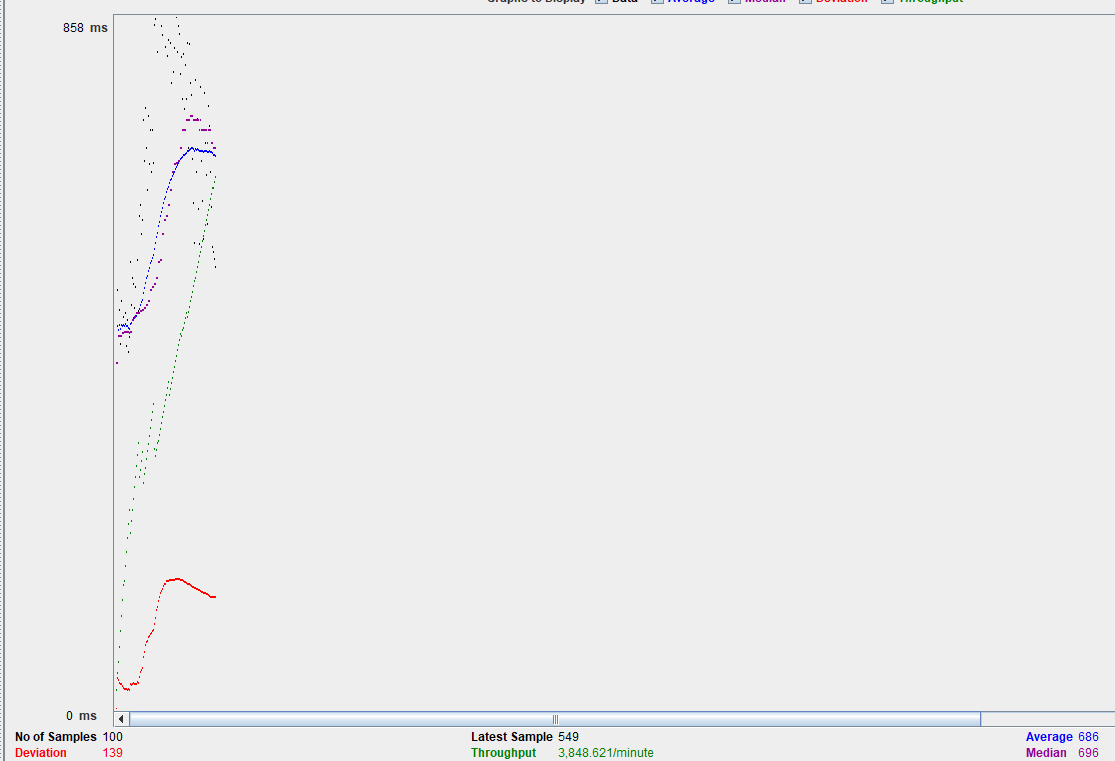
**Ans.**

**Architecture Diagram**

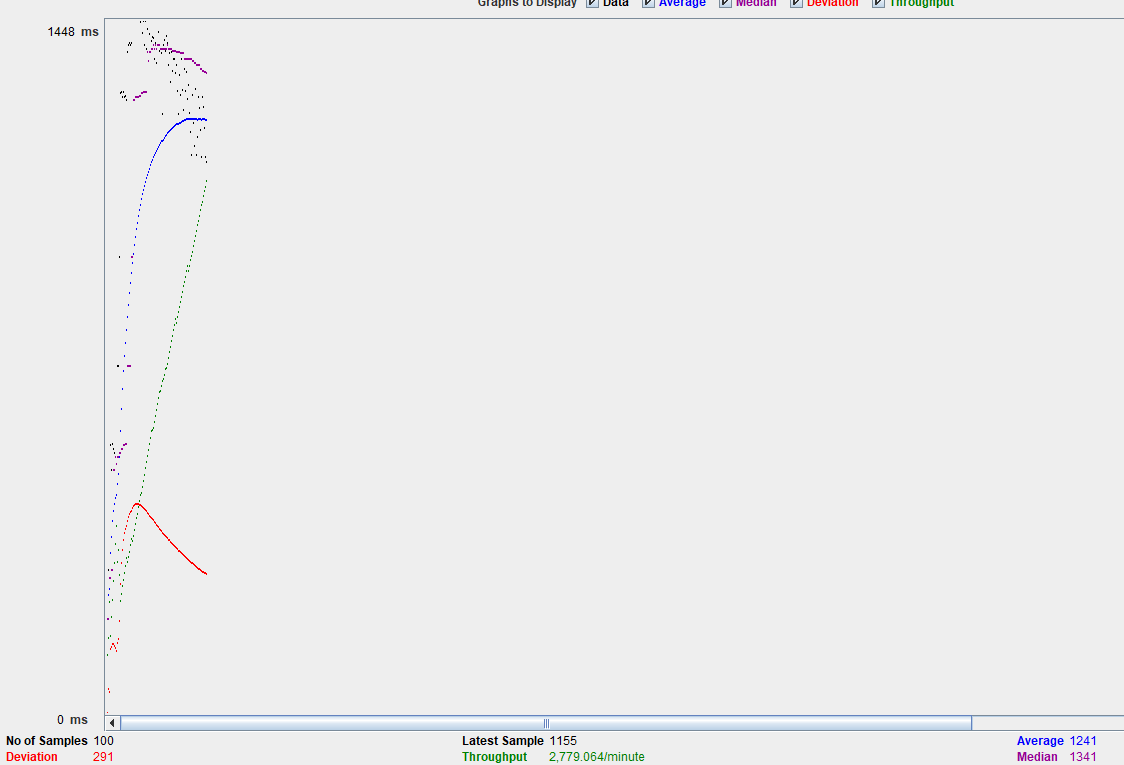


**Jmeter Testing**

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



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



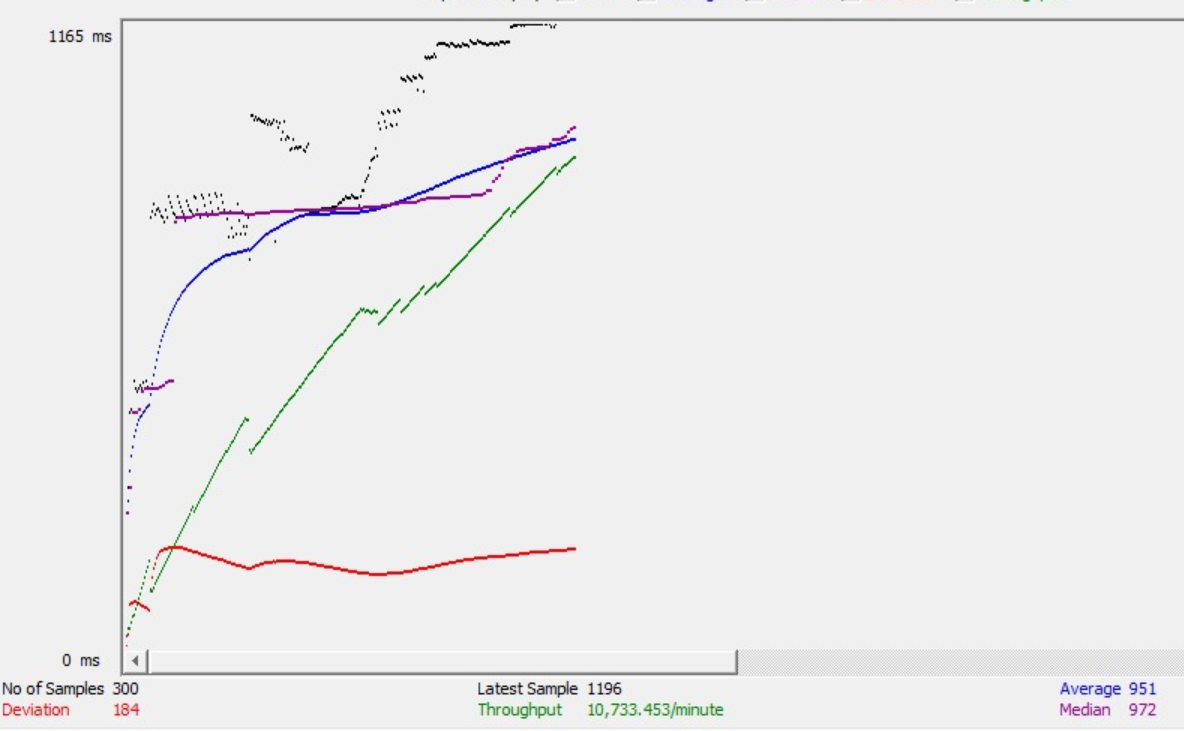
1. **200 concurrent users skill/allSkills - with connection pooling (Average 842)**



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



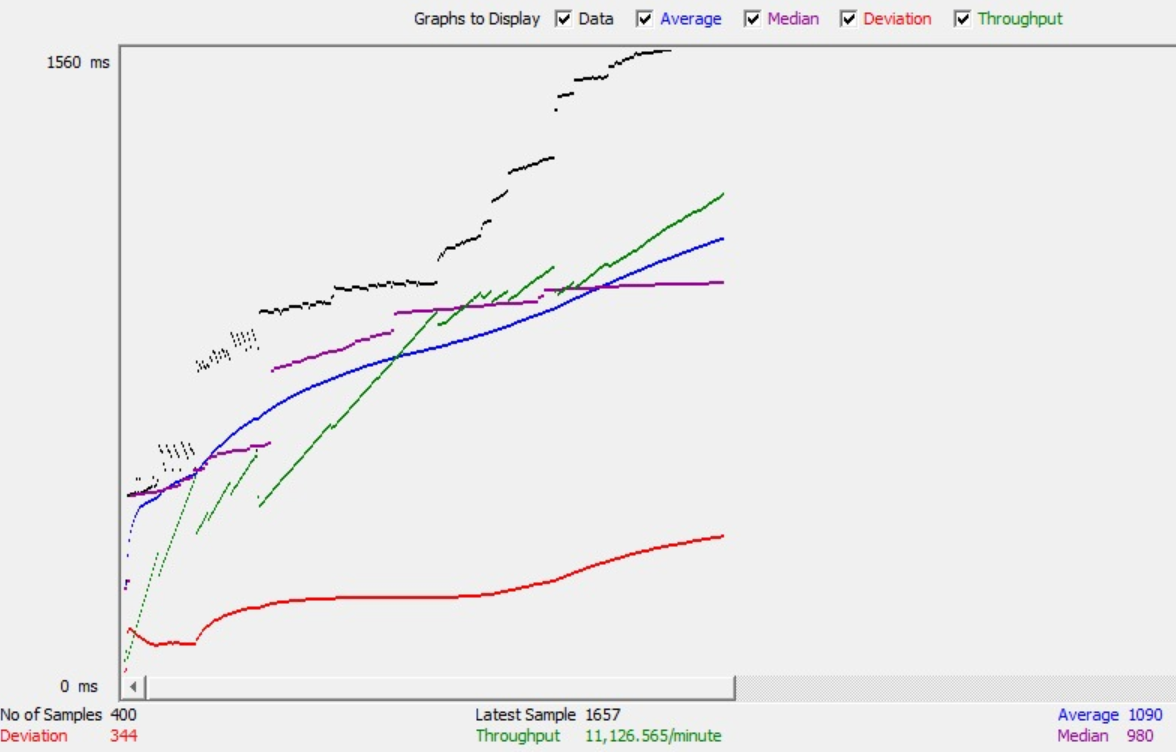
1. **300 concurrent users skill/allSkills - with connection pooling (Average 951)**



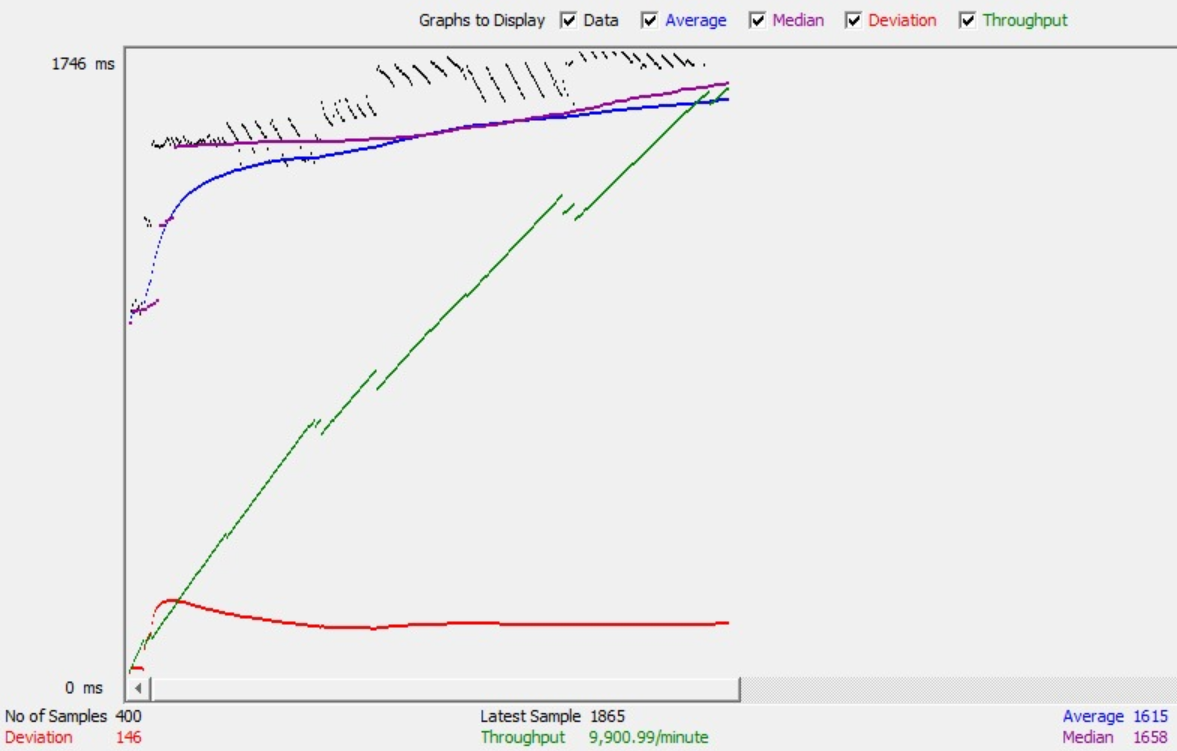
1. **300 concurrent users skill/allSkills - without connection pooling (Average 1431)**



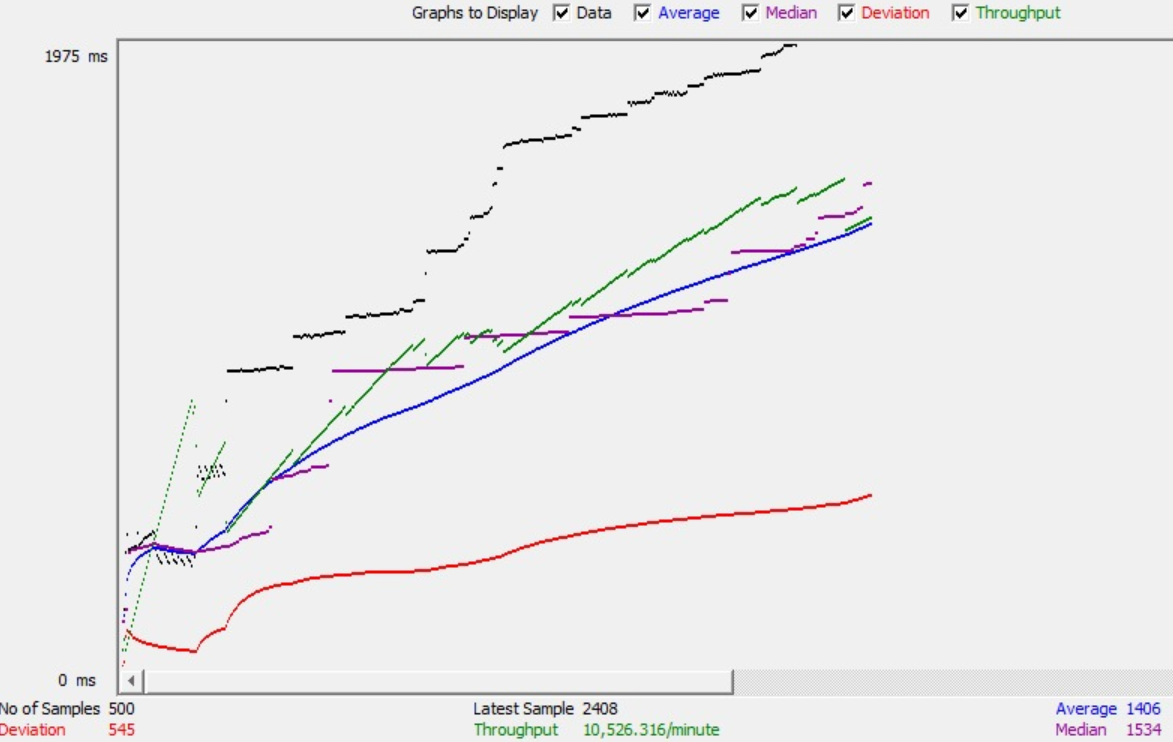
1. **400 concurrent users skill/allSkills - with connection pooling (Average 1090)**



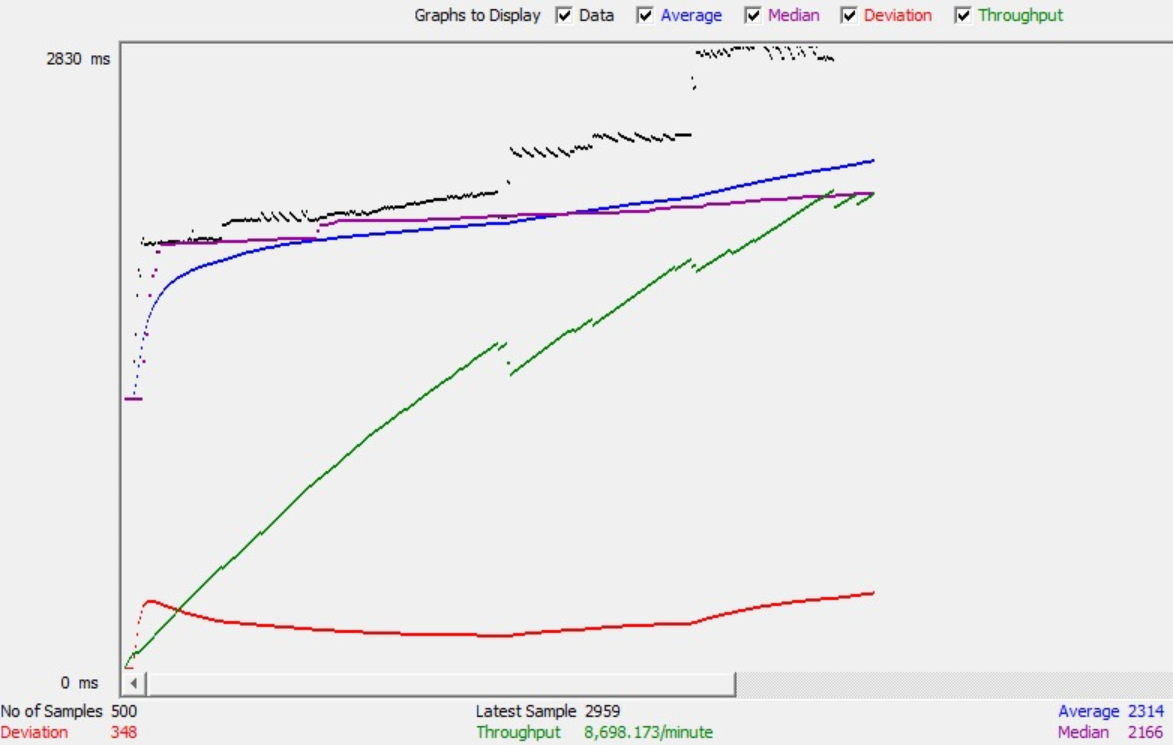
1. **400 concurrent users skill/allSkills - without connection pooling (Average 1615)**



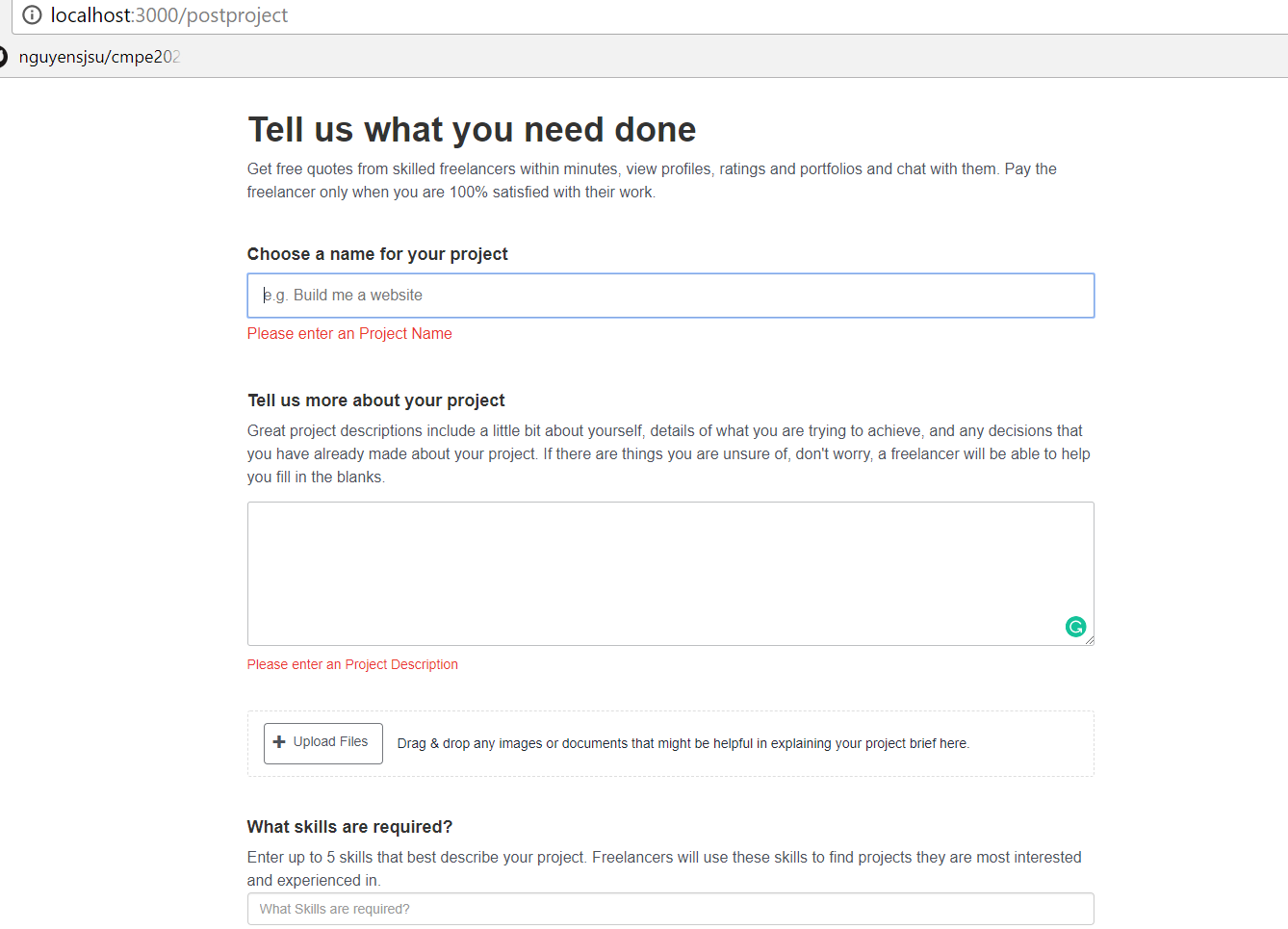
1. **500 concurrent users skill/allSkills - with connection pooling (Average 1406)**

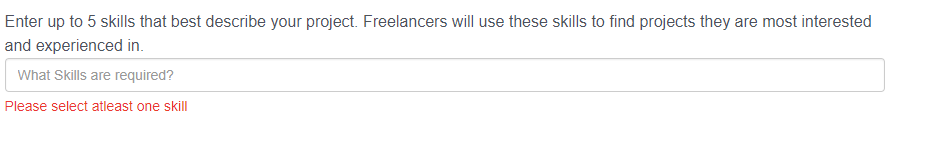


1. **500 concurrent users skill/allSkills - without connection pooling (Average 2314)**

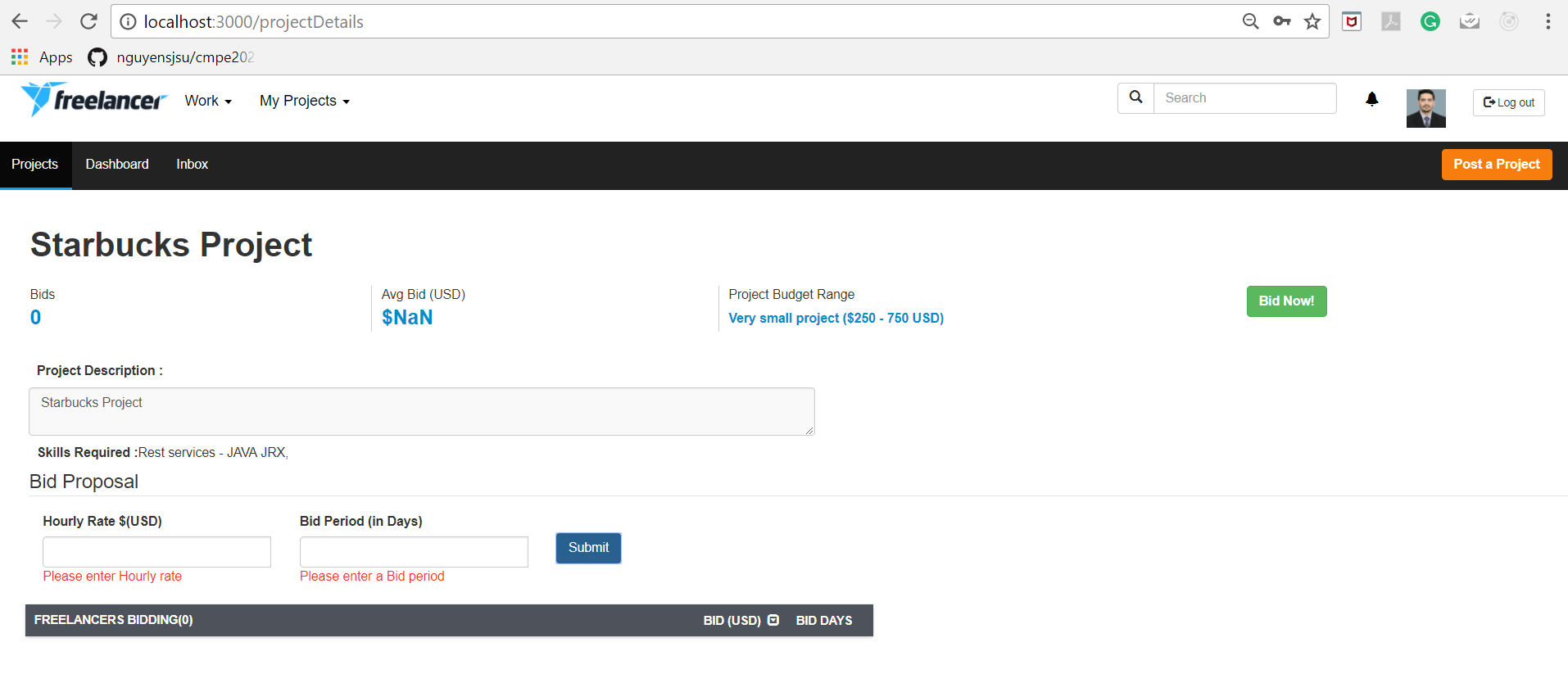


1. **Post Project validations**

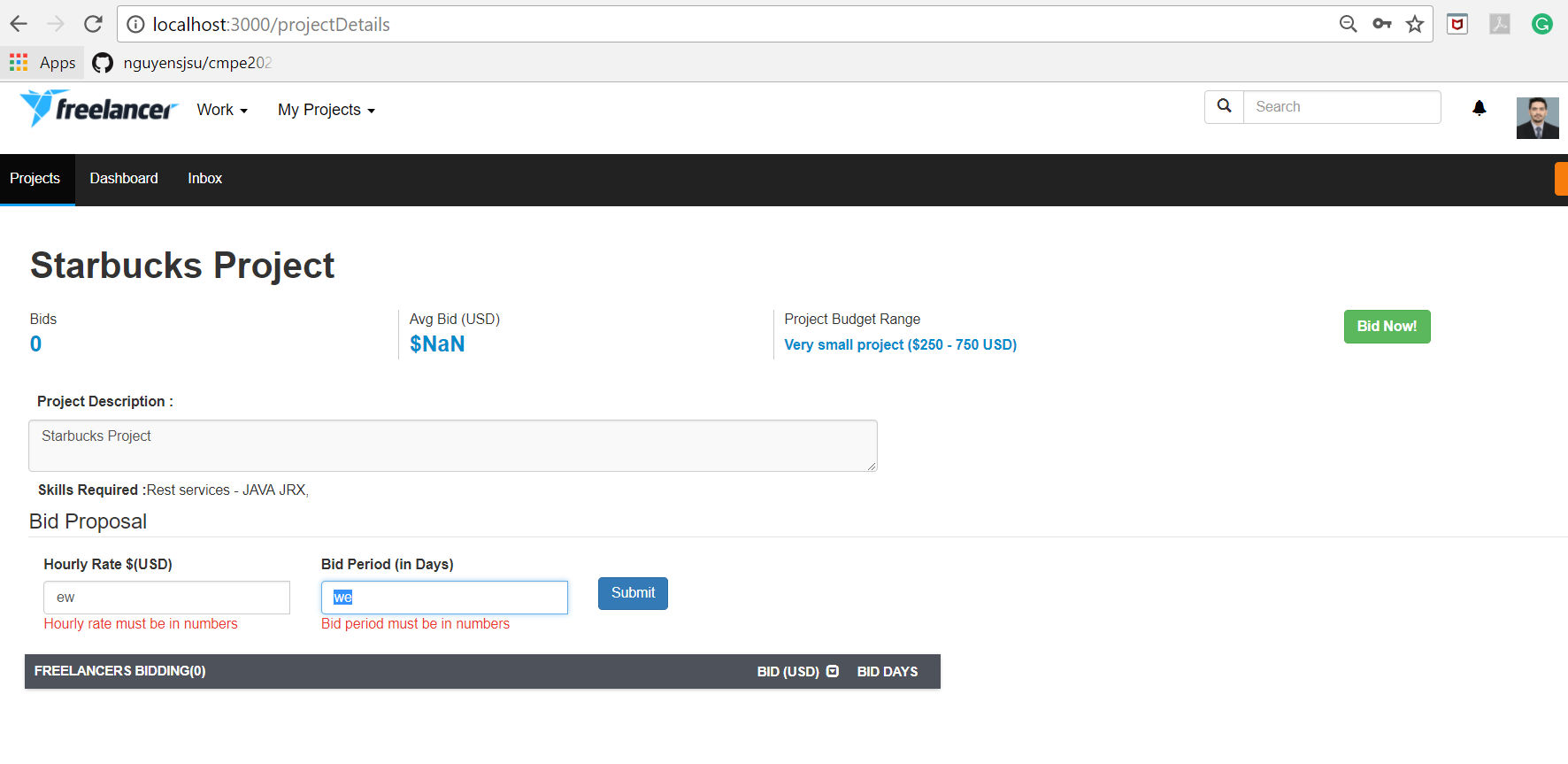




1. **Bid Form Validations ( check whether value has been entered)**

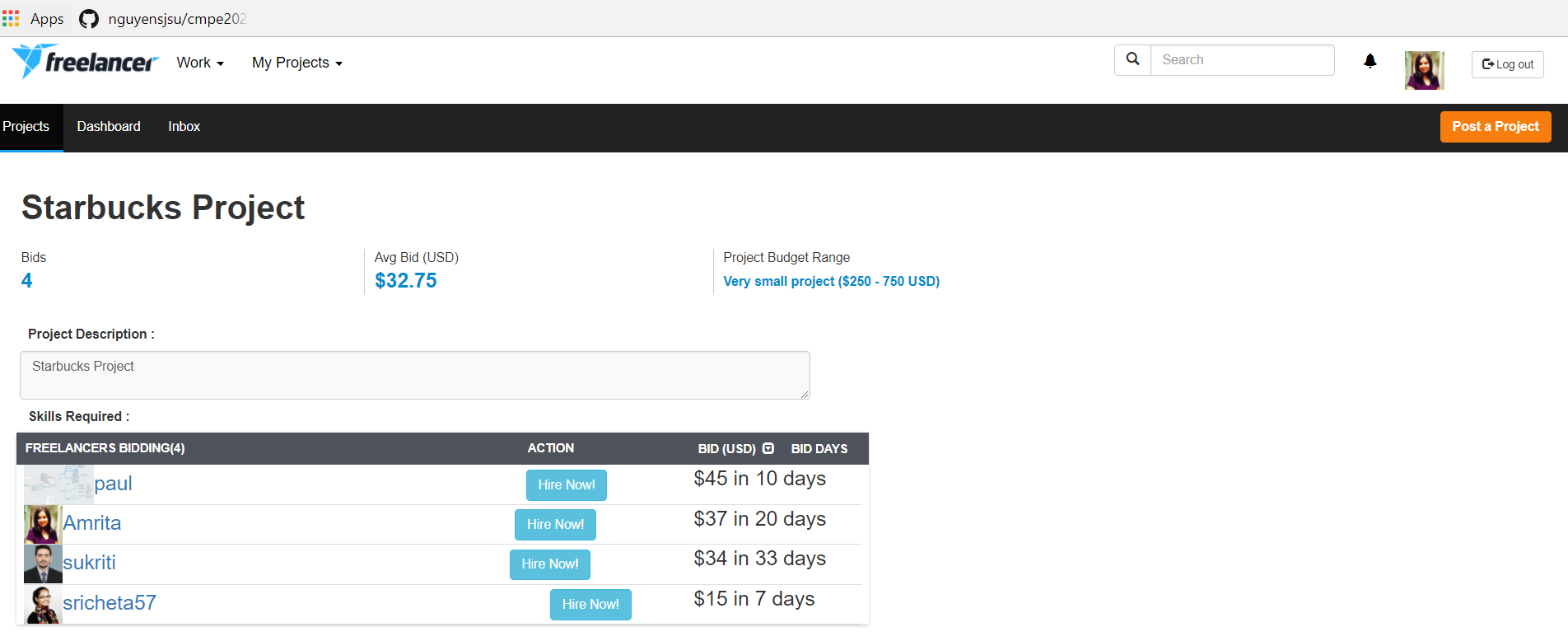


1. **Bid Form Validations ( check whether value is numeric or not)**

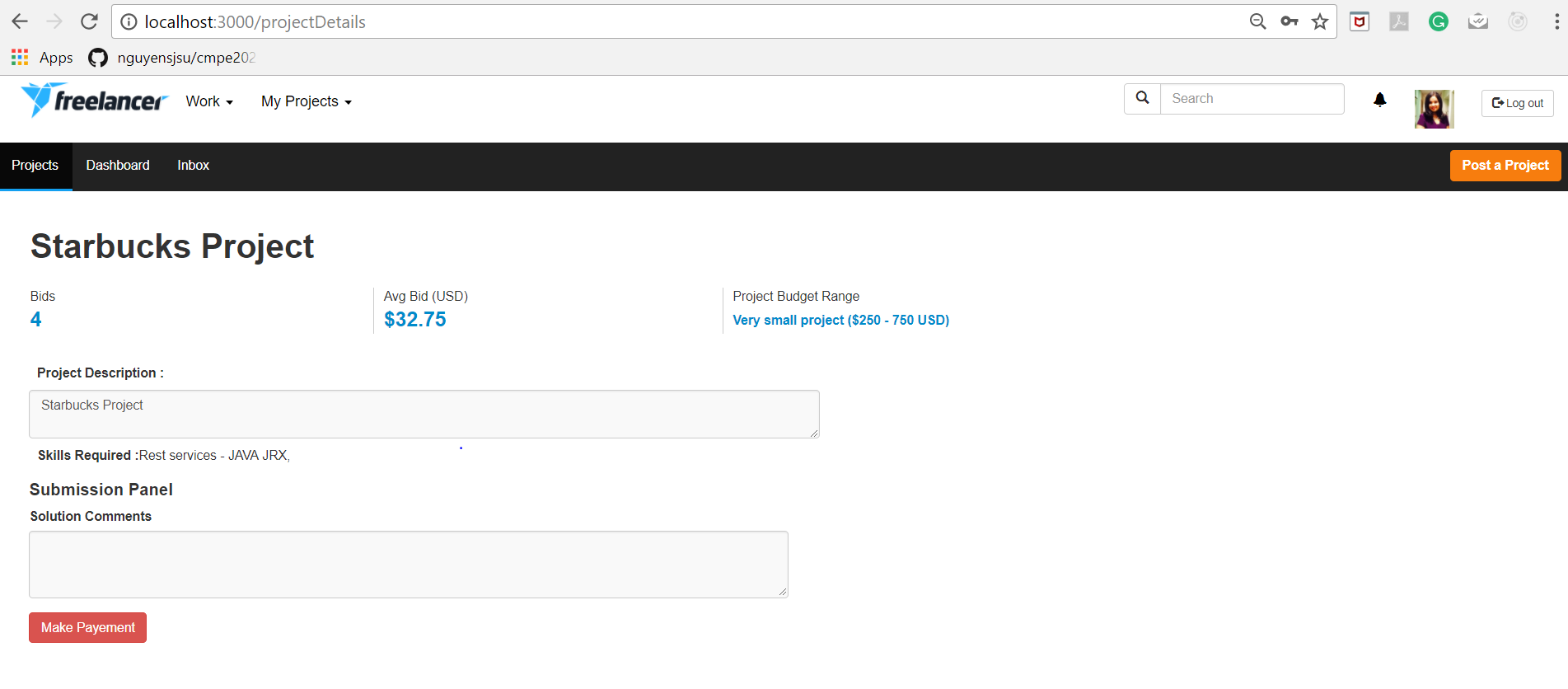


1. **Sorting on user bid list**

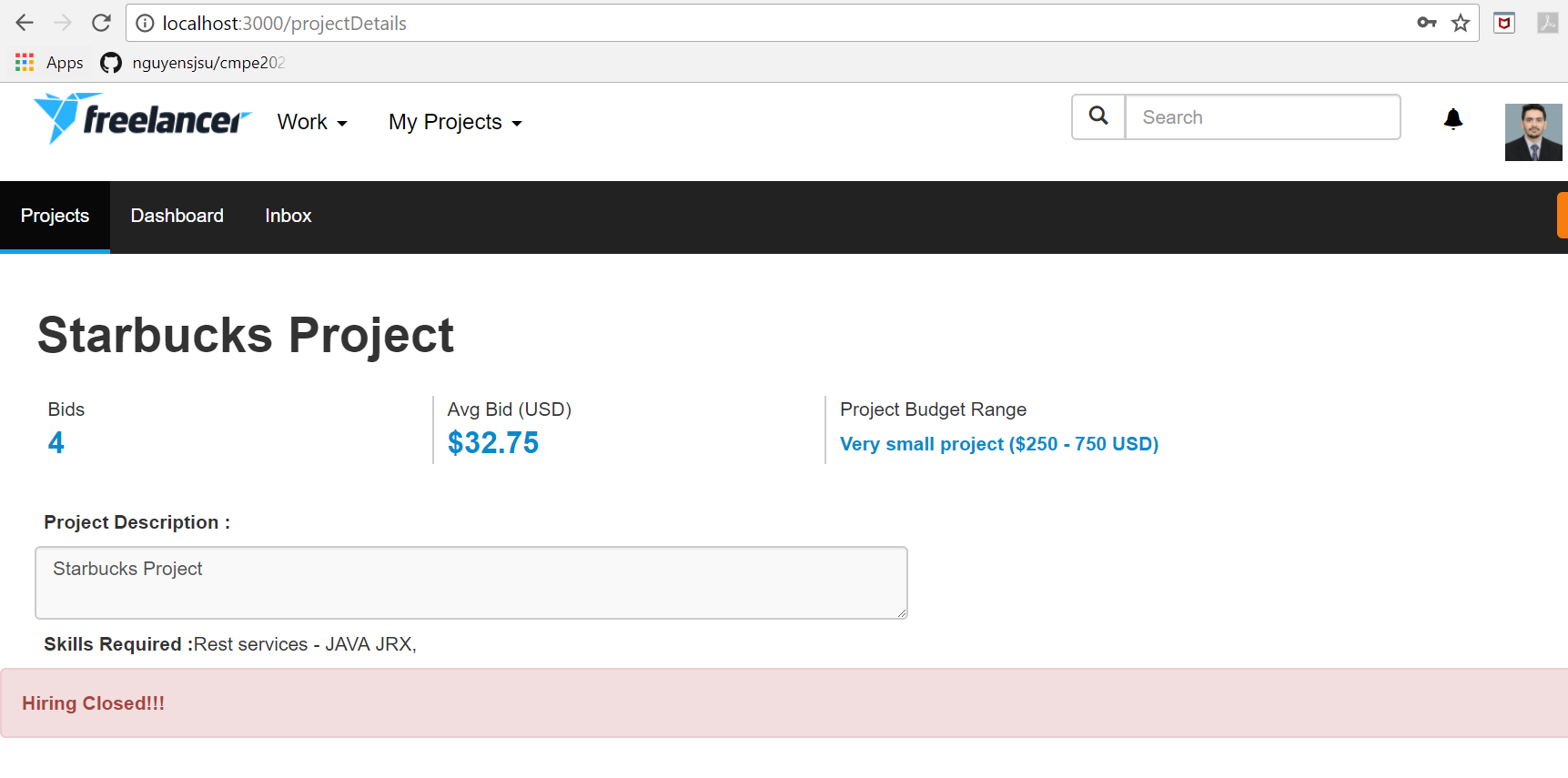




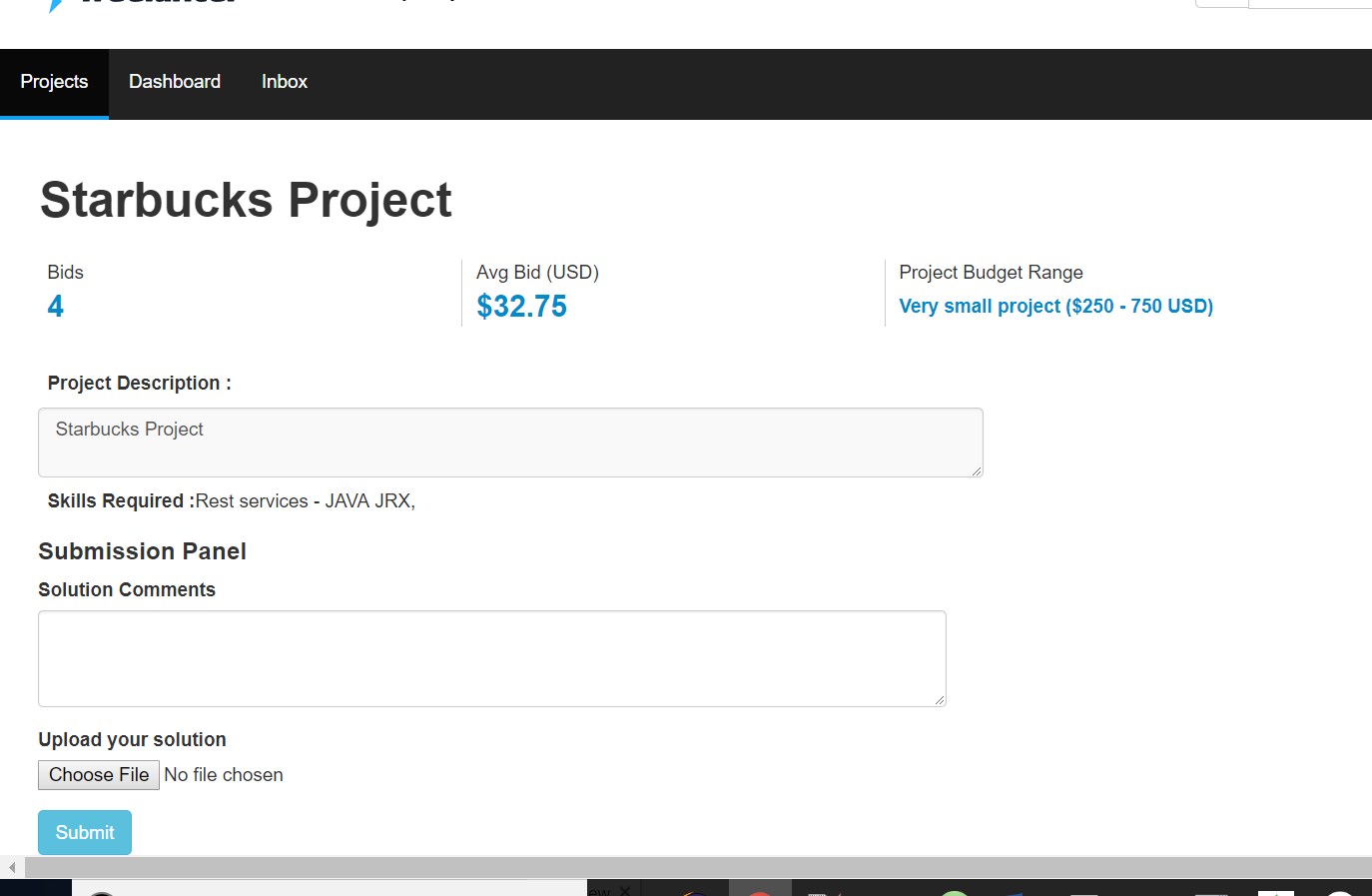
1. **After click on Hire Now button, Make Payment screen is visible to Employer.**

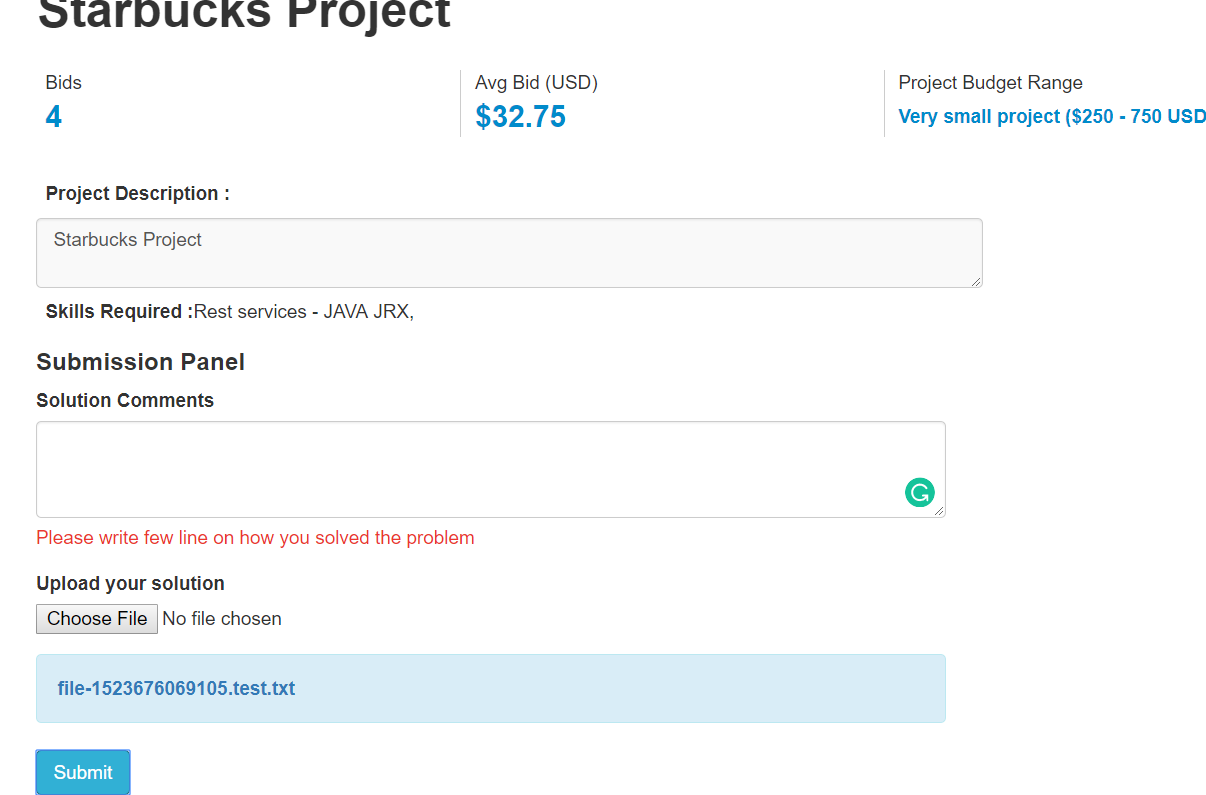


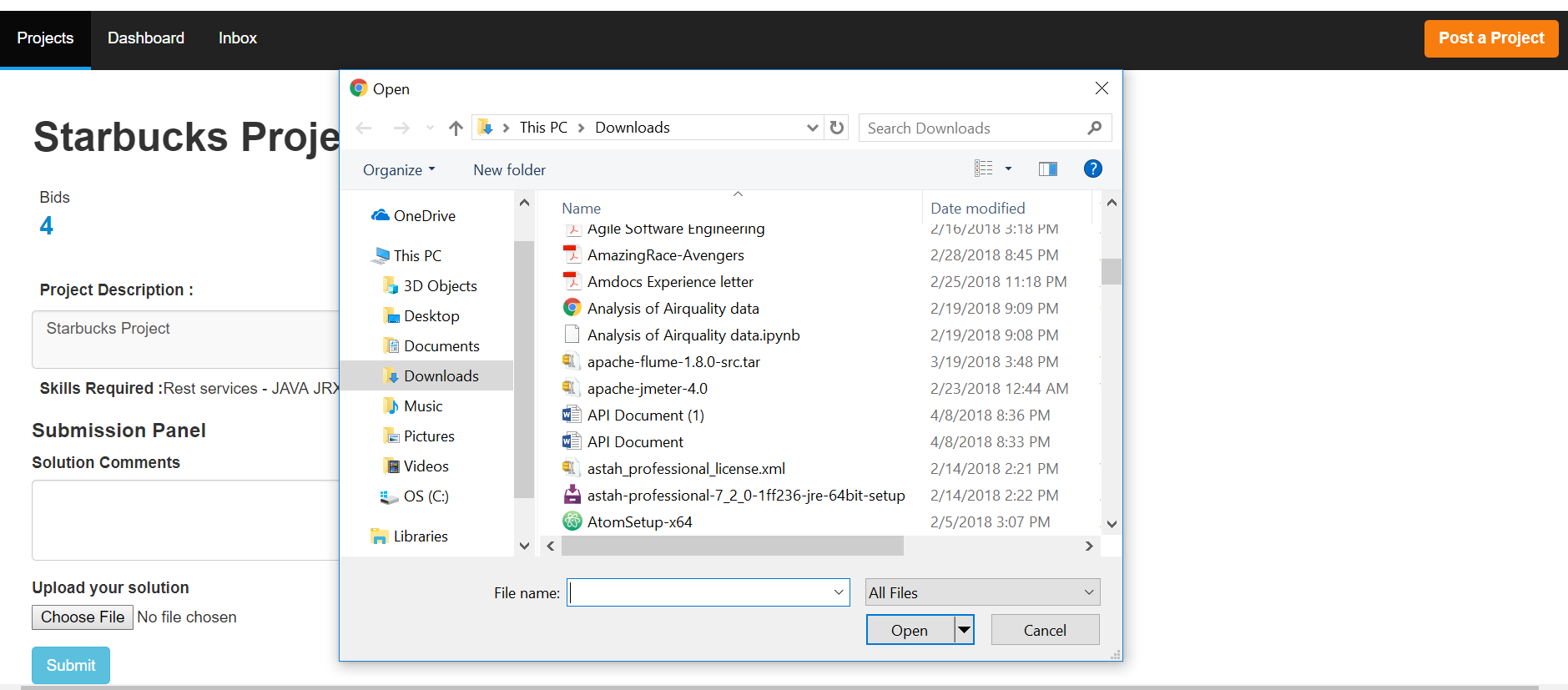
1. **If a freelancer is not hired, then for then this screen is visible**

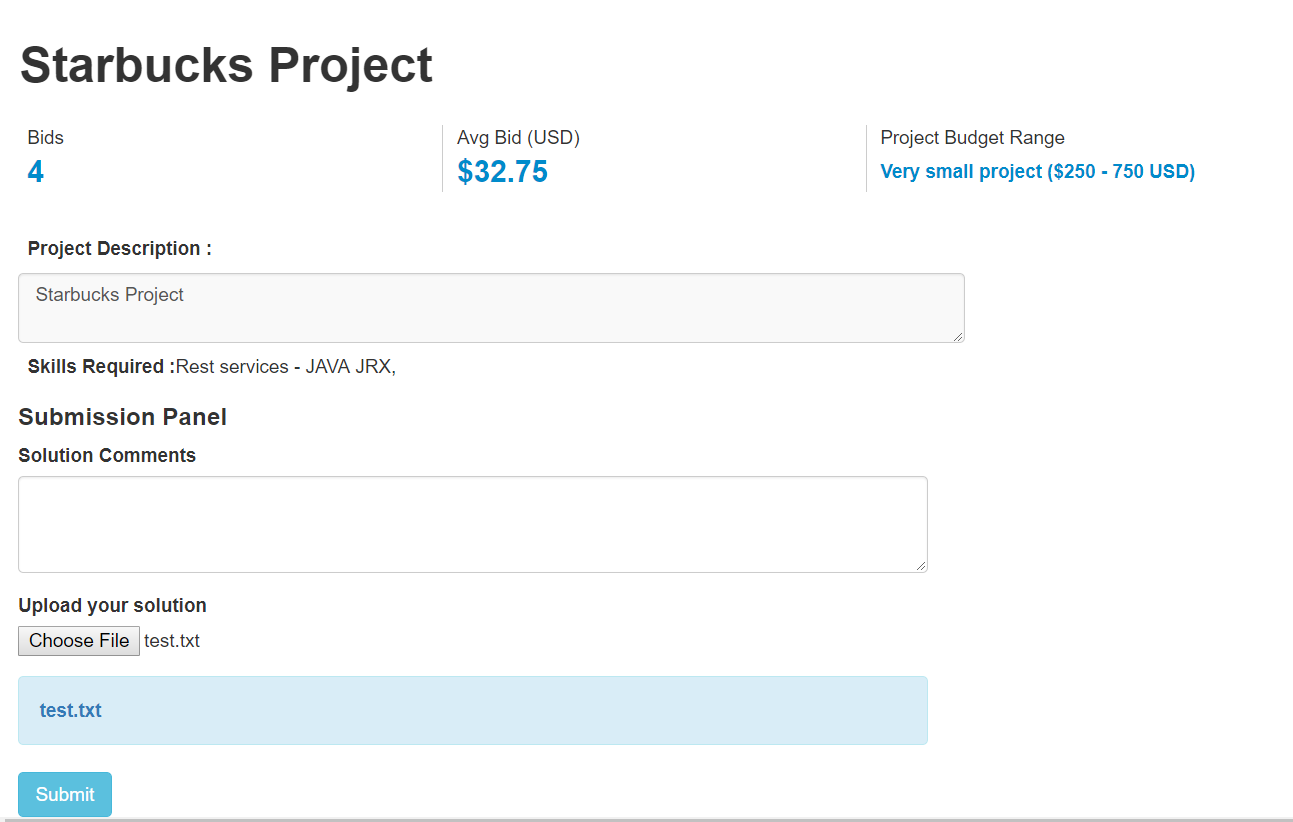


1. **If a freelancer is hired, the for them submission screen is visible (Validations included )**

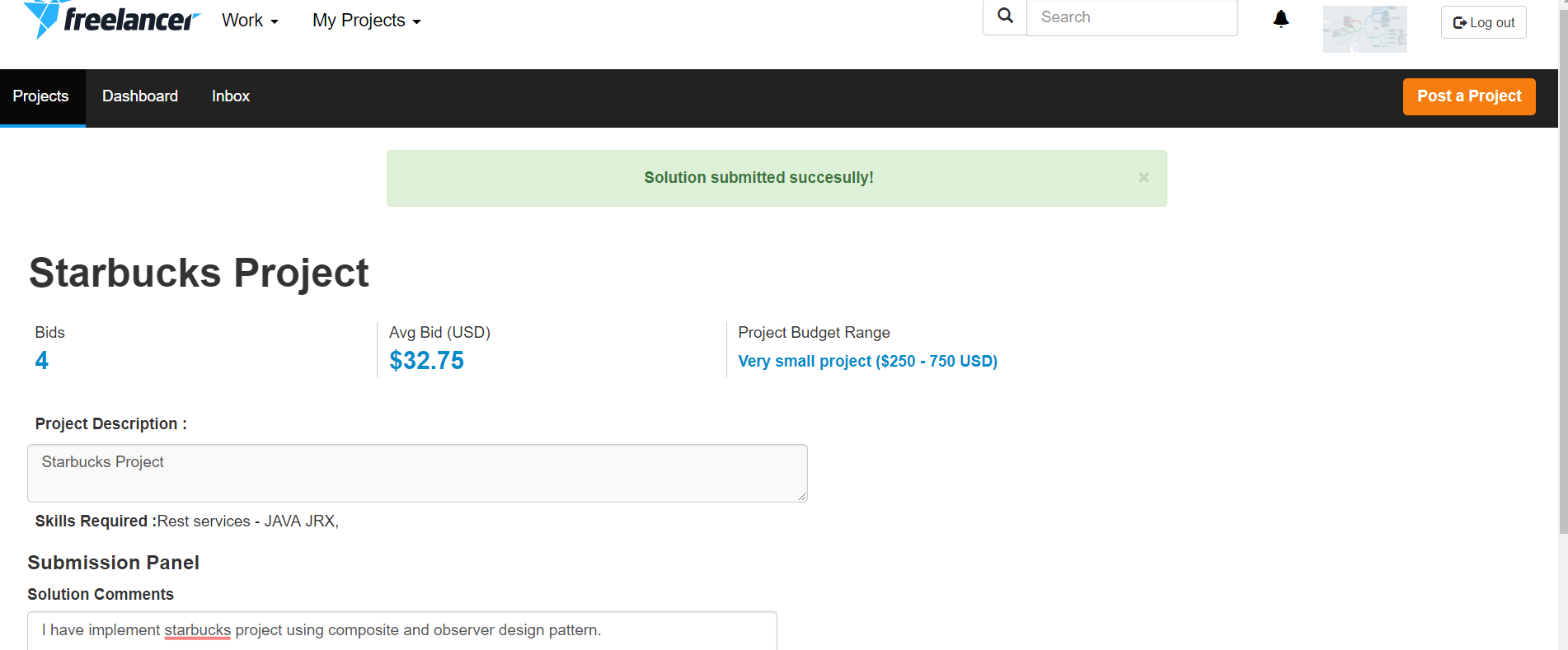




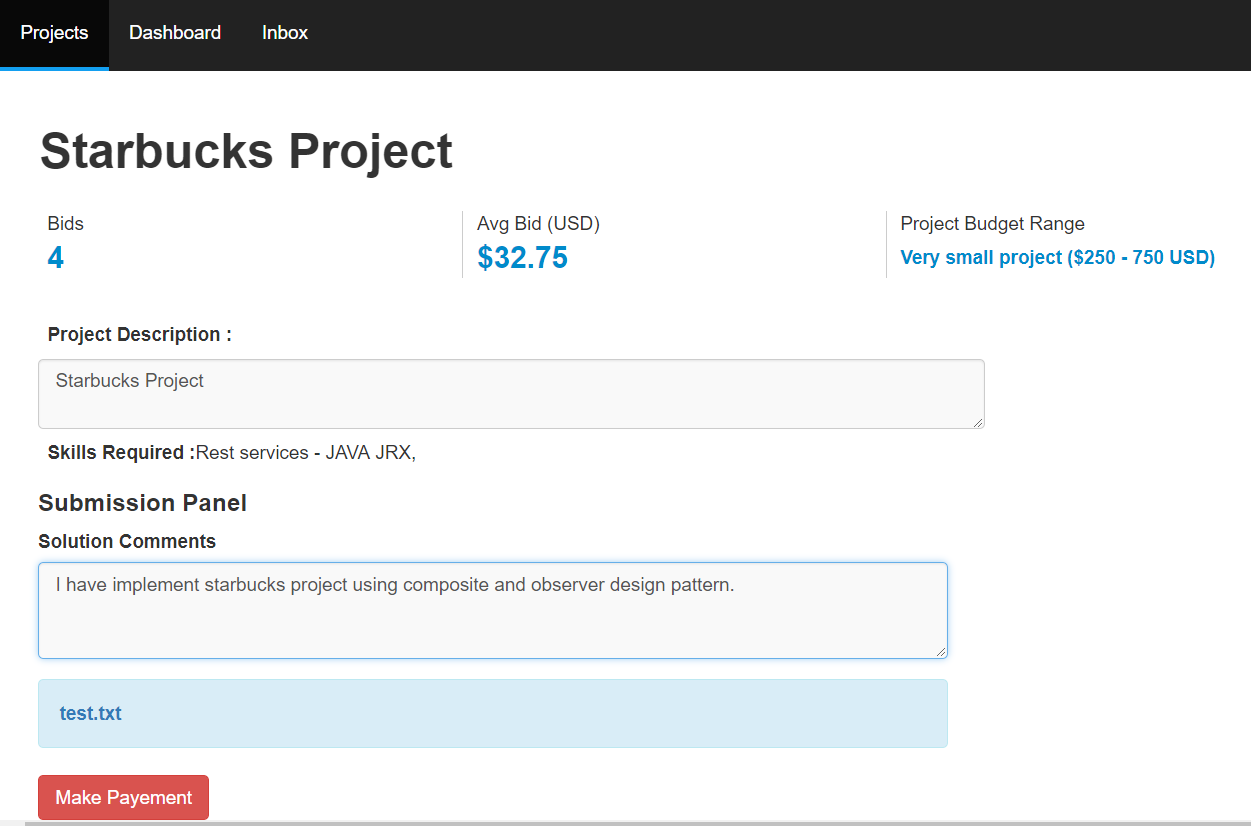


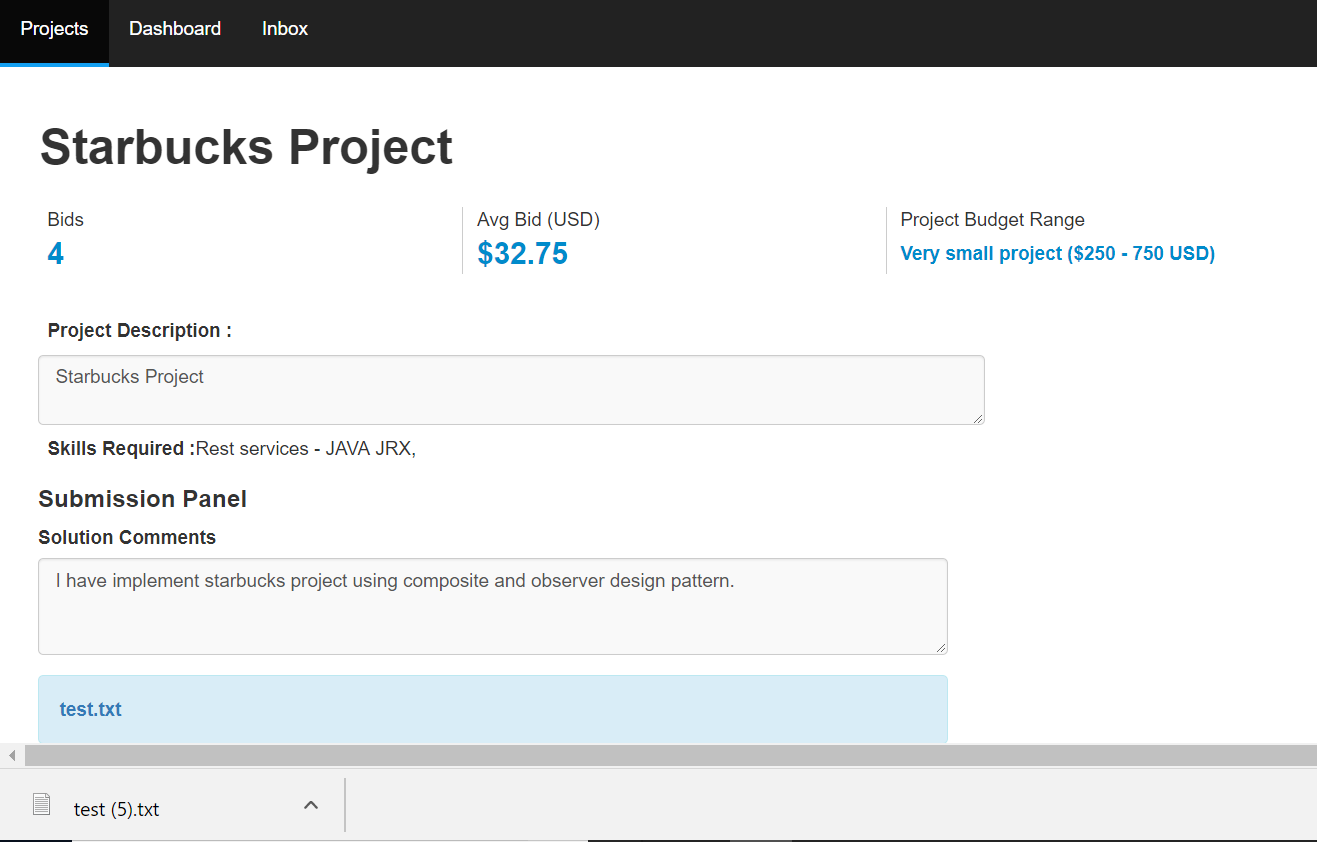


1. **Submit solution to Employer**

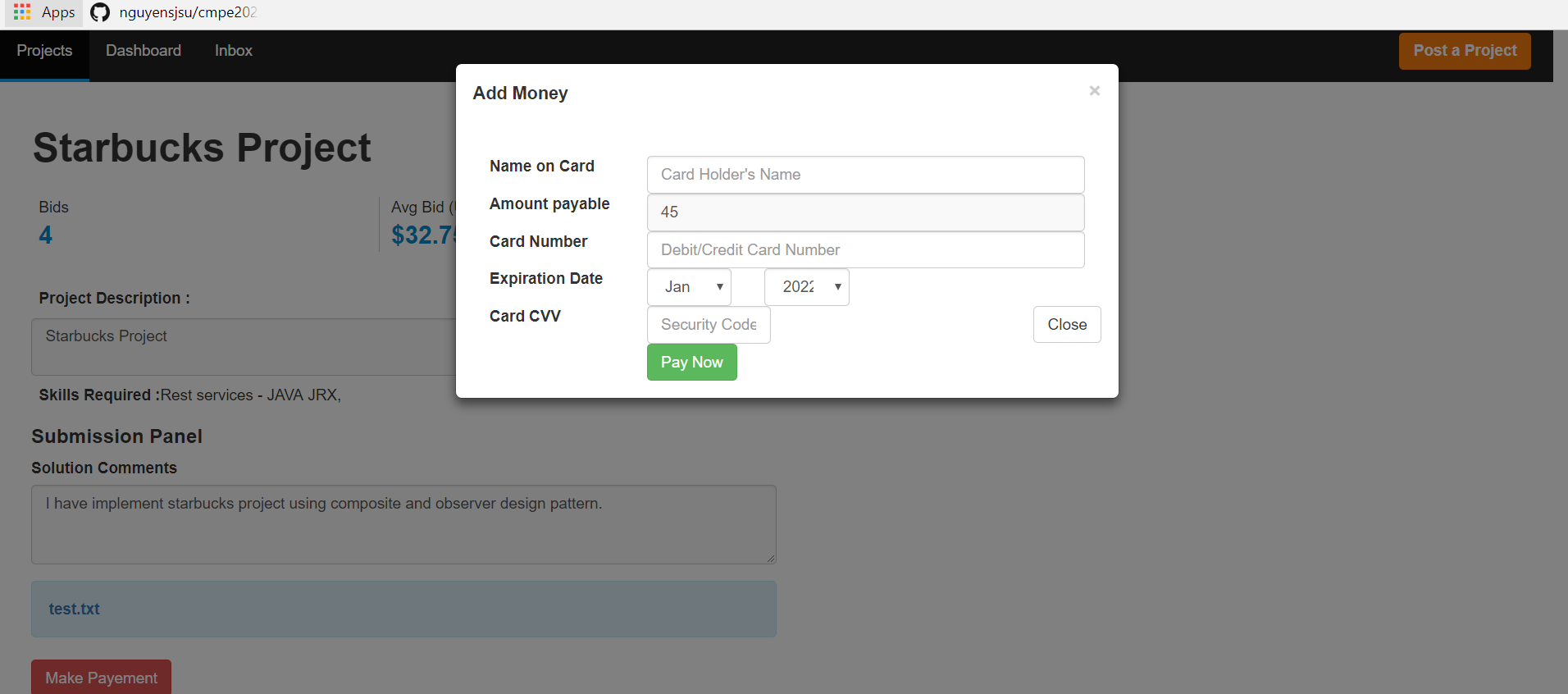


1. **Project Details screen for Employer after solution is submitted ( Files downloadable )**

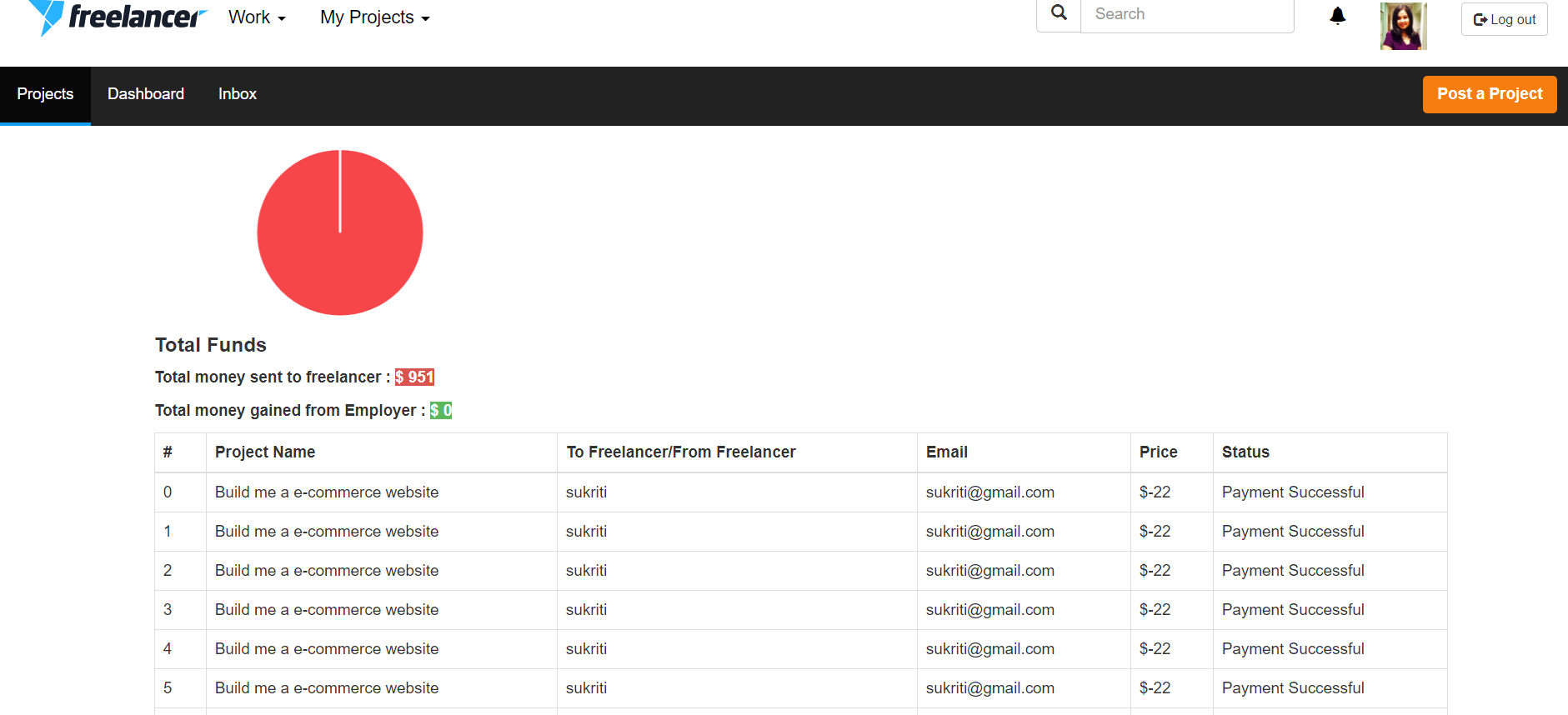




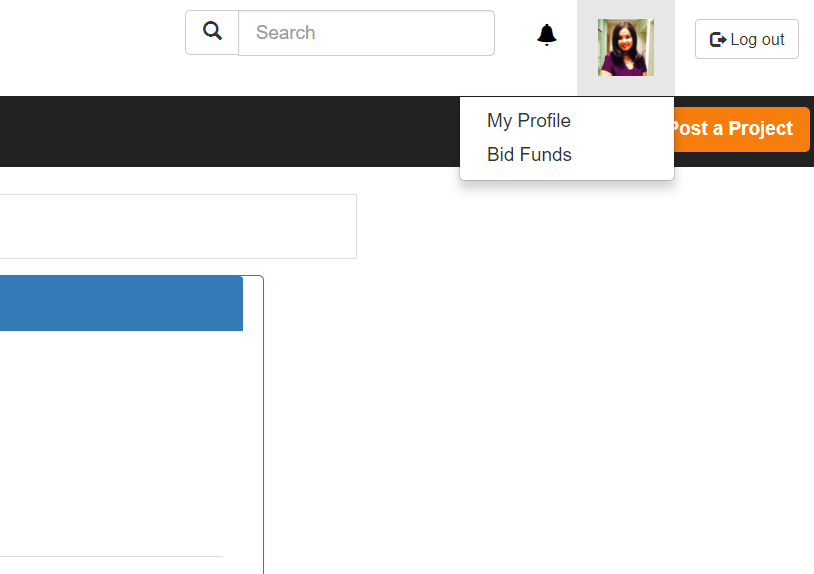
1. **Make Payment Modal with amount payable in disabled state ( same as bid price)**



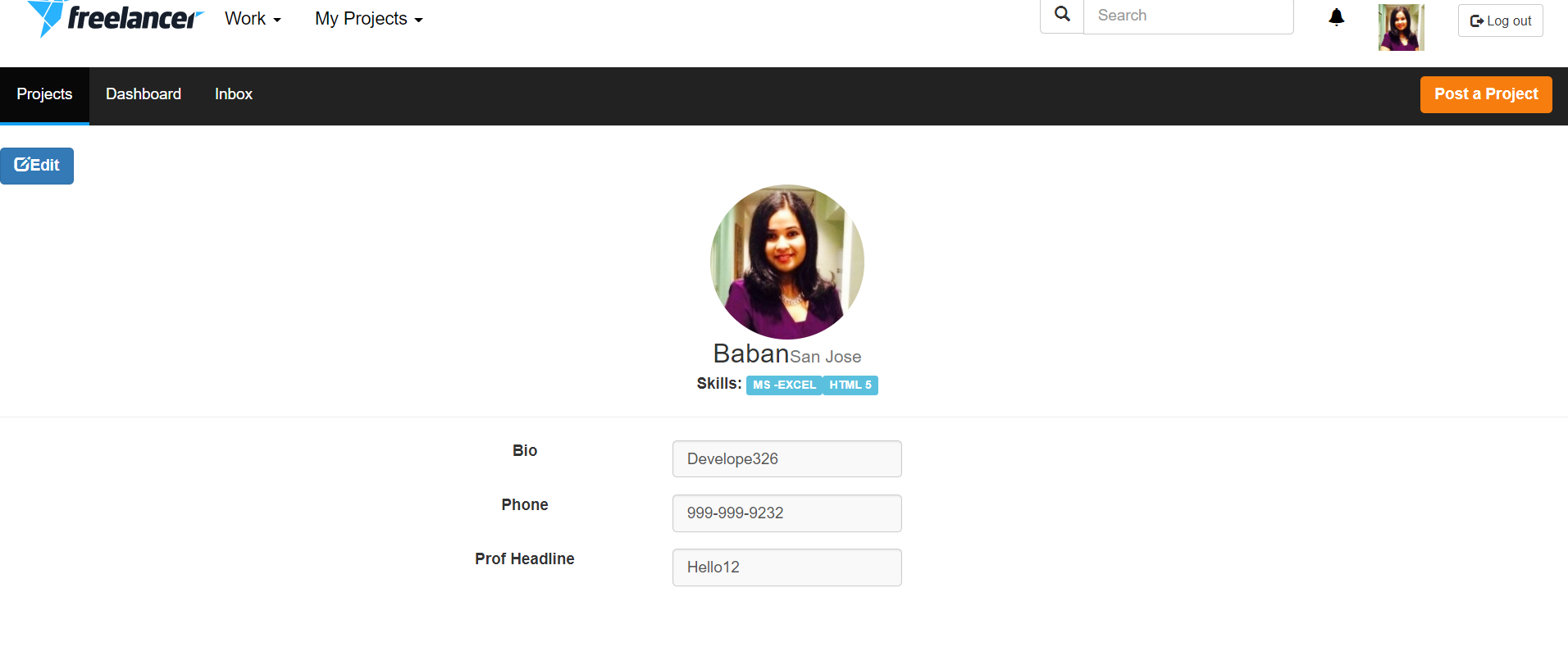
1. **Click on Pay Now button navigates to Transaction history page**



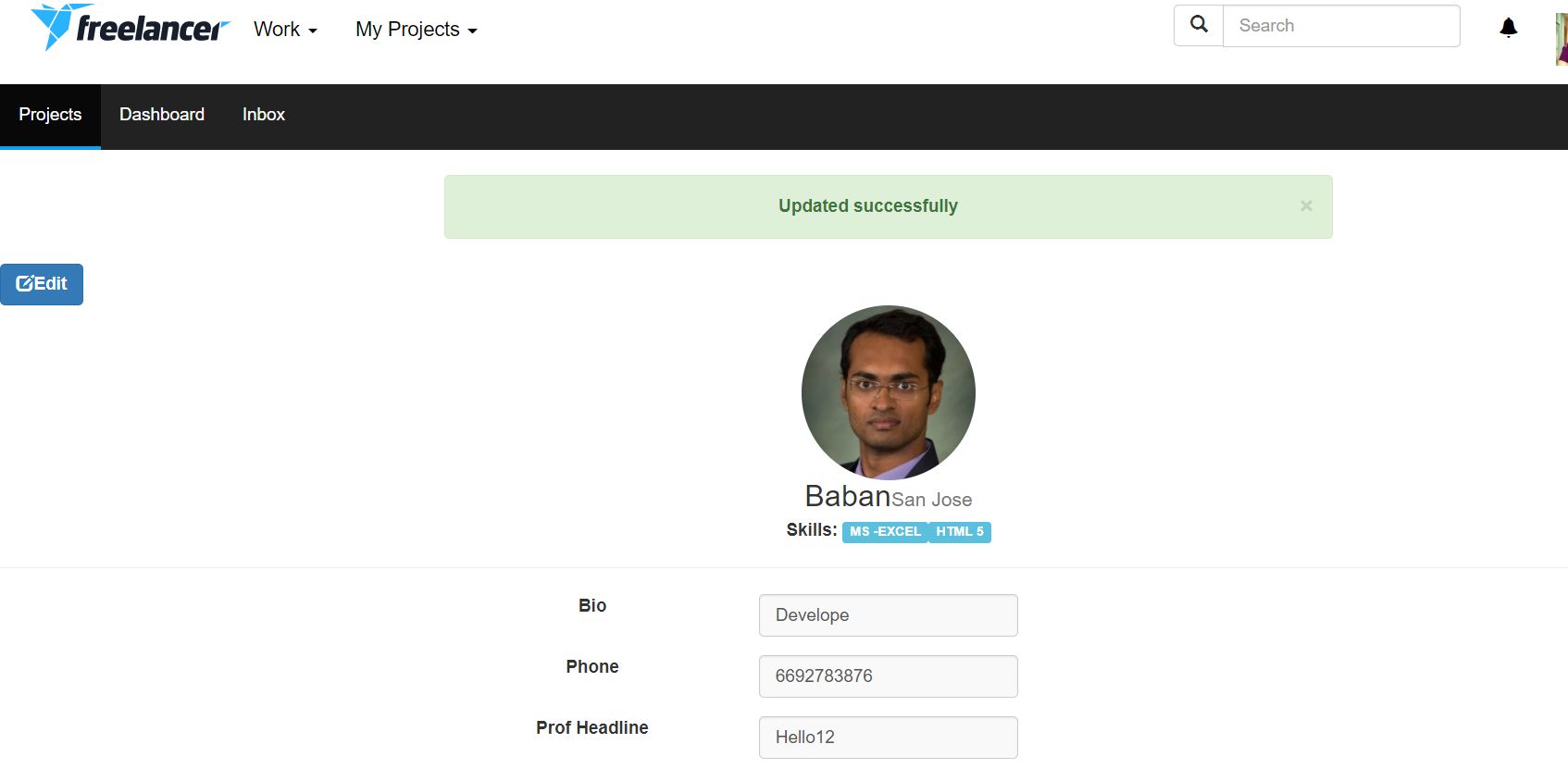
1. **User has option to see this funds from navbar menu as well.**



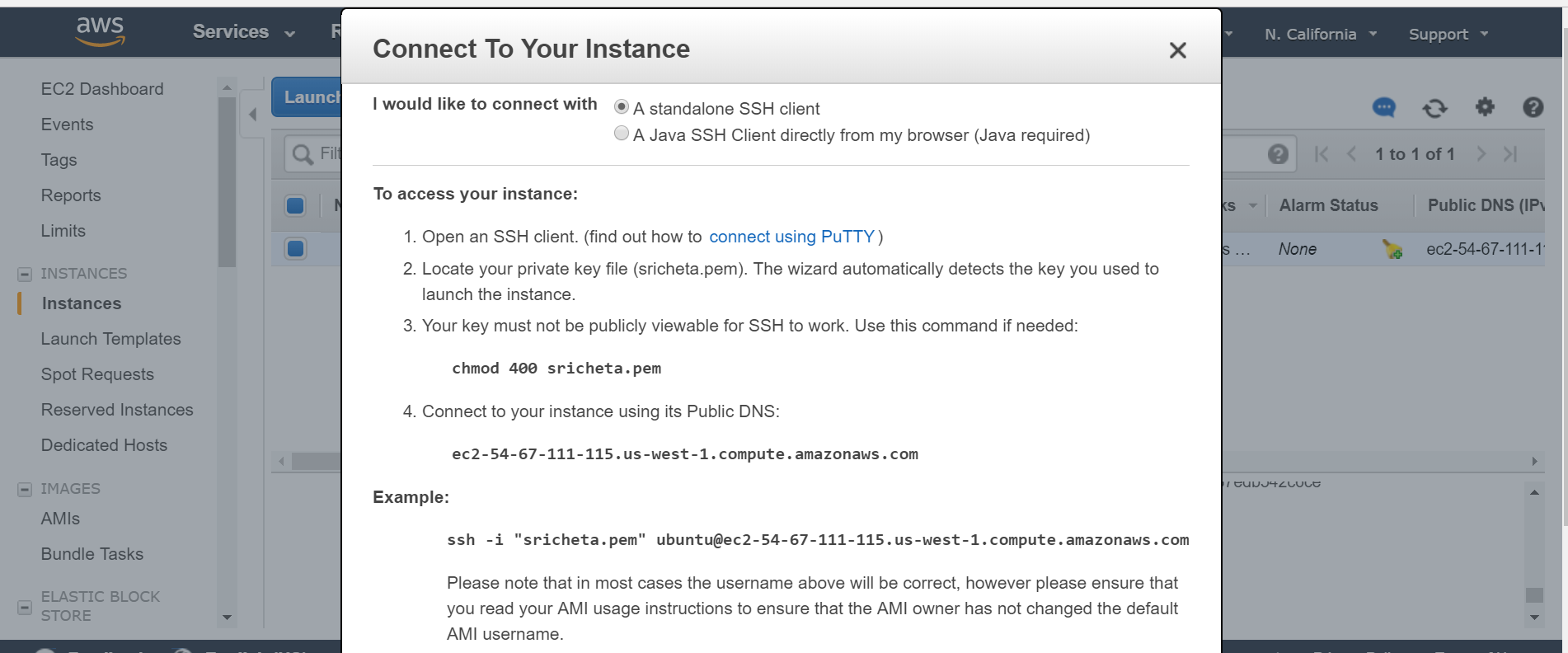
1. **View Profile**



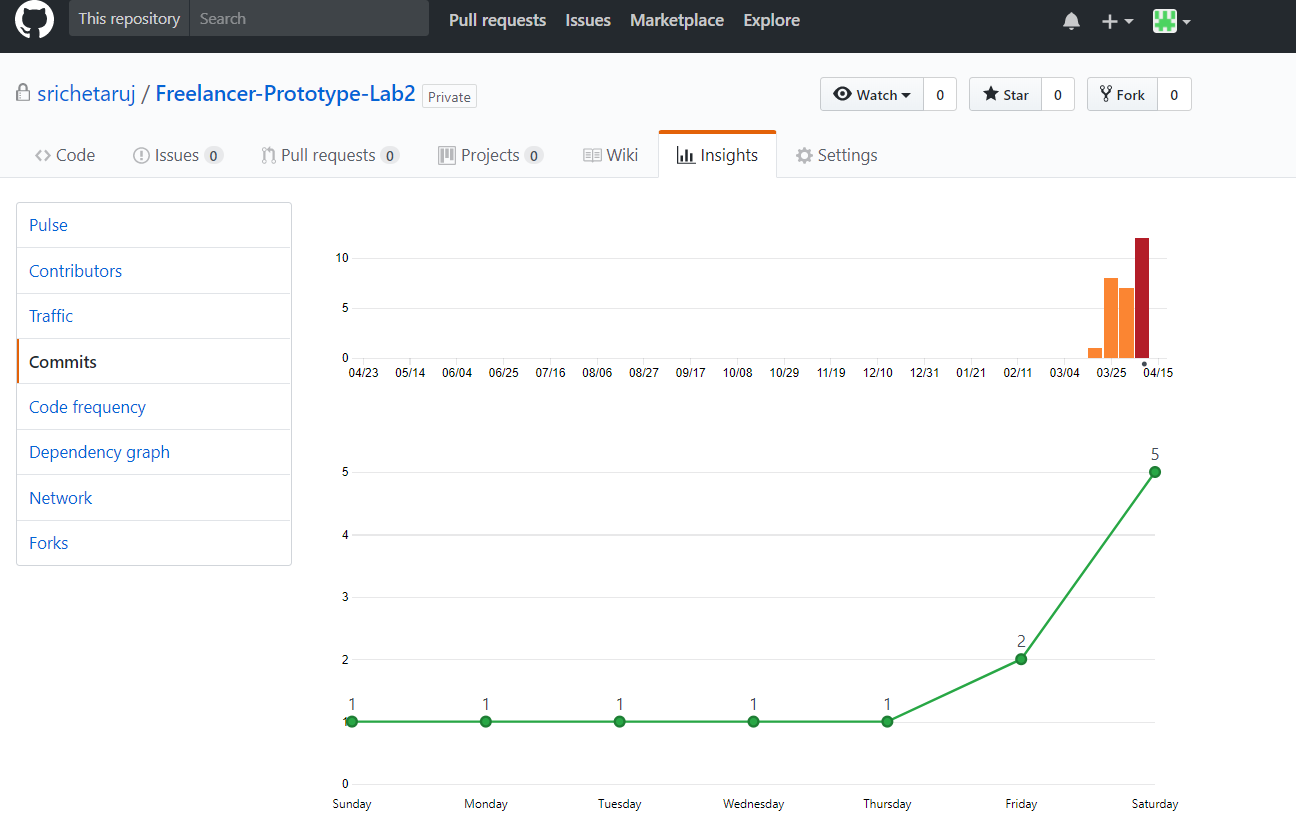
1. **Update Profile**



**Cloud hosting**



**Git commit history**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **/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 |

**Mocha Testing**

