San Jose State University Computer Engineering Department



CMPE 275 – Enterprise Application Development

**PROJECT 2**

**PROJECT FREEDOOR**

Instructor: Prof. John Gash

Date of Submission: 10th Dec, 2014

**Submitted By:**

Minu Theresa Thomas

Kinjal Gandhi

Deepthi Babbula

Shivadeepthi Toopran

Laxmi Phalak

**Table of Contents**

1. **INTRODUCTION 3**
2. **SCOPE AND OBJECTIVES 3**
3. **ASSUMPTIONS 4**
4. **FUTURE ENHANCEMENTS 4**
5. **SYSTEM ARCHITECTURE 5**
6. **DATABASE SCHEMA 8**
7. **PROJECT DESCRIPTION AND SCREENSHOTS 9**
8. **Load Balancing 9**
9. **REST API calls 11**
10. **Testing 26**
11. **CHALLENGES 31**

**1. Introduction**

What is barter system?

A barter system is an old method of exchange. This system has been used for centuries and long before money was invented. People exchanged services and goods for other services and goods in return.

How to make an offer?

* + For which item I am making an offer?
  + What Date
  + Quantity
  + Waiting time
  + List of the things I am going to trade
  + Accept /Reject Offer

The goal of this project is to create an online barter system that is like a community of sorts. The idea is to exchange skills or goods and materials in order to support the community. The final goal is to create a scalable network of systems that is loosely connected through REST APIs.

Each team will have their own cluster of systems that will handle requests. The architecture of the cluster will be such that one server will be solely dedicated in load balancing and the rest of the servers will be servicing the request. Each team will have their own database in order to handle the data.

All the teams will be able to connect their team clusters with other teams to form a single large barter community where any one team can send a request to any other team. The request will be made in the format of the REST API which is mutually agreed upon by all teams in the community.

REST service was decided upon as the best approach in order to create a scalable network. We have used Node.js as the technology and MySQL for the database.

In our cluster, we have implemented the Round Robin algorithm for load balancing using the http-proxy module. All the servers in our cluster use a common shared database which is in the cloud for improved accessibility.

In the following sections, we will explain the scope of our project and each of the features we implemented in detail. We will also cover all the challenges we faced and the assumptions made in the project which we will solve as a part of future enhancements.

**2. Scope and Objectives**

The current scope of project includes:

1. Create an online barter system cluster which will receive the request from the client and perform the requested action on the data.
2. One server is given the task of load balancing and dividing the work between the other servers.
3. The load balancing is done in a Round Robin fashion and each request is serviced by the selected server.
4. The cluster uses a shared database to store the data, which is on the cloud for easy accessibility.
5. Each individual cluster can be joined to other clusters to create a massive network which will service the client request.
6. Mutually agreed upon REST APIs will be used to communicate with other team clusters.
7. The user can create Categories and Products inside those Categories.
8. A user can make an offer on another user’s product in which he will specify what he is willing to exchange in return for the product.
9. The seller can either approve or reject the buyer’s request.
10. The offer history is maintained in order to track any change made in the offer.
11. The buyer and seller can also add comments on the offer in order to communicate with each other.

**3. Assumptions**

Due to system limitations and a local environment without enterprise level components, we have assumed certain things in the design of our system that ensure the service to the client.

These assumptions are:

1. All the servers need to be up and running. We do not have any mechanism to handle server failure. If the server processing the request goes down, the request will be lost.
2. The load balancer server should always be up and running. There is only one node (server) handling the load balancing. This server is the sole point of contact with the user. So if the load balancer server goes down, there is no way to handle the requests.
3. We assume that user will know the Leader IP and port.
4. The load balancer server will not do any processing. Only the Slave server nodes will process the requests. The only thing the load balancer will decide is to which server the request should go for processing based on the Round Robin method.

**4. Future Enhancements**

Due to time limitation there are certain things we could not implement in our system. As a part of future enhancements we will try to implement the following things:

1. **Implement load balancing using the Seaport module:**

Currently we are using the http-proxy module in npm to implement the load balancing. In the future we will implement load balancing using the Seaport module.

The Seaport module has the heartbeat functionality which lets us know which servers are up and running so that we avoid sending requests to a server which is not up and running. This will provide our system with a mechanism to handle server failure.

1. **Leader elections for load balancer server:**

Currently if the load balancer server goes down, we lose the sole point of contact with the user. Also none of the other servers can handle load balancing.

Hence in the future, we would be implementing the logic to conduct a leader election to select the server which will perform the load balancing and will be the point of contact with the user.

**5. System Architecture**

The architecture of the Barter System consists of five main components, namely:

1. REST API call
2. Intra Cluster Shared MySQL DB on the cloud
3. Intra Cluster Load Balancer Server
4. Intra Cluster Slave Servers
5. Inter Cluster Barter System

The architecture can be depicted as below –

**Shared MySQL Db**

**(Cloud)**

**REST API call**

1. **REST API call:**

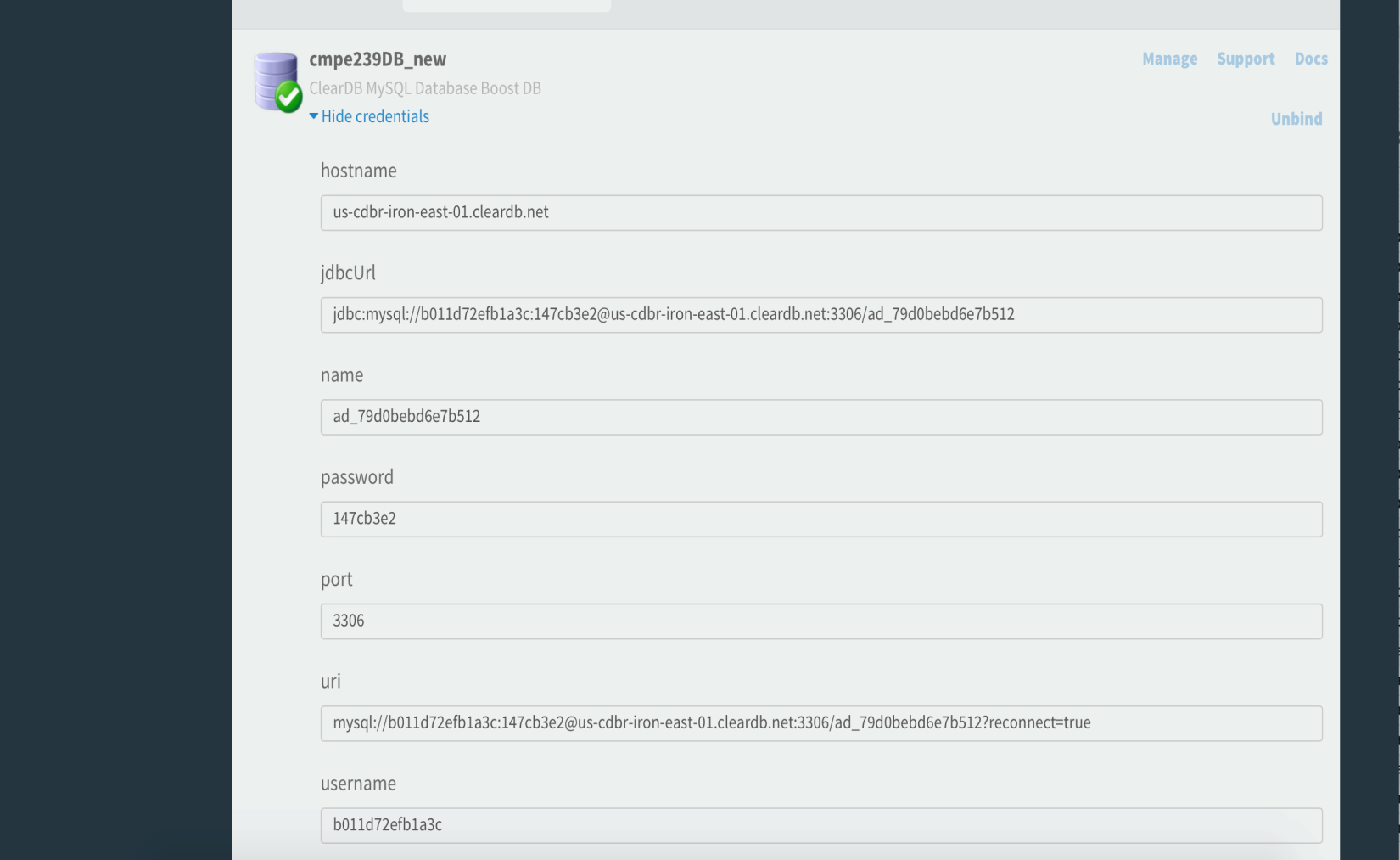
The REST API calls have been mutually agreed upon in the community and only requests pertaining to that format will be processed. The REST API calls are made through a REST client.

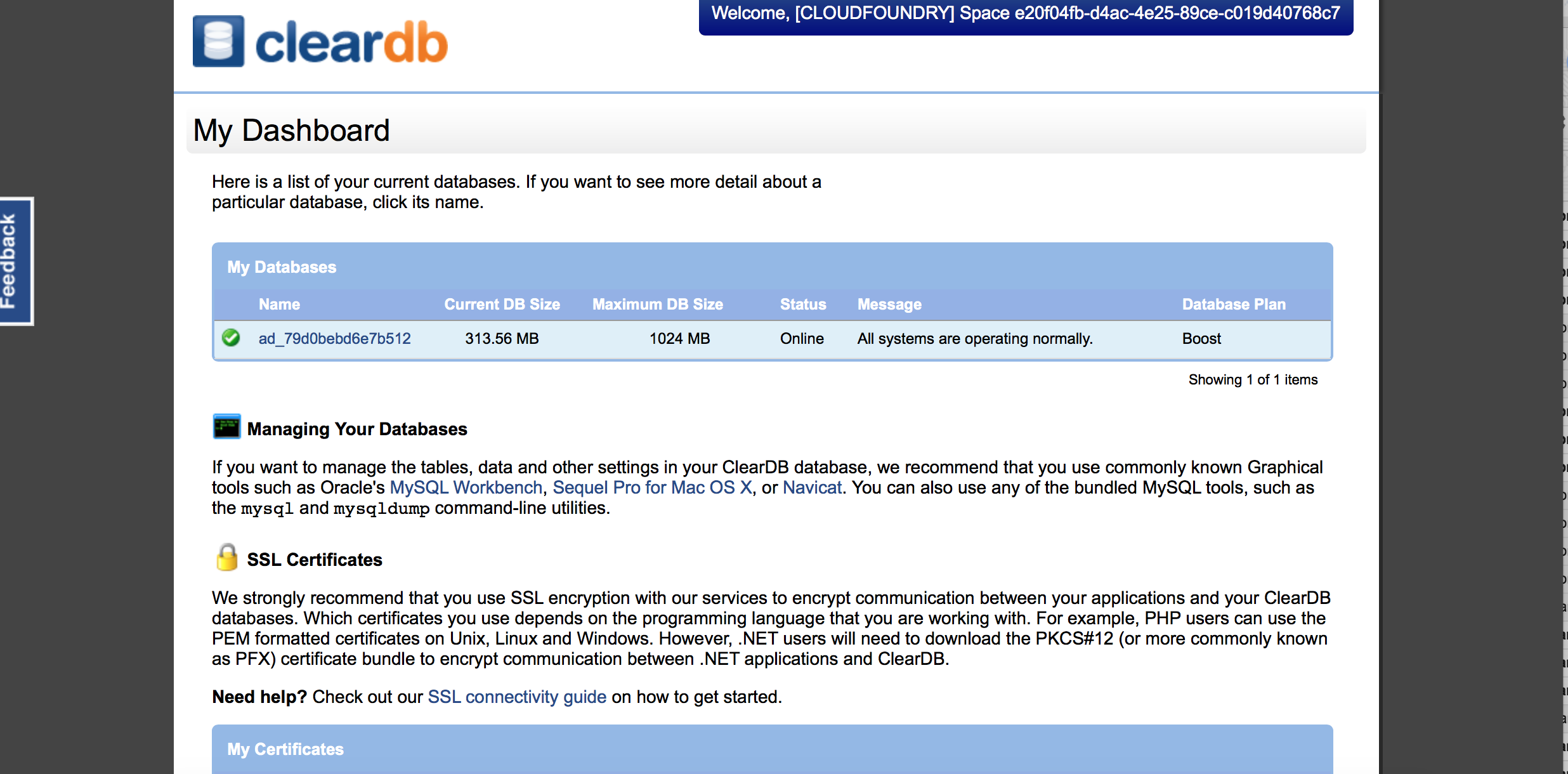
1. **Intra Cluster Shared MySQL DB on the cloud:**

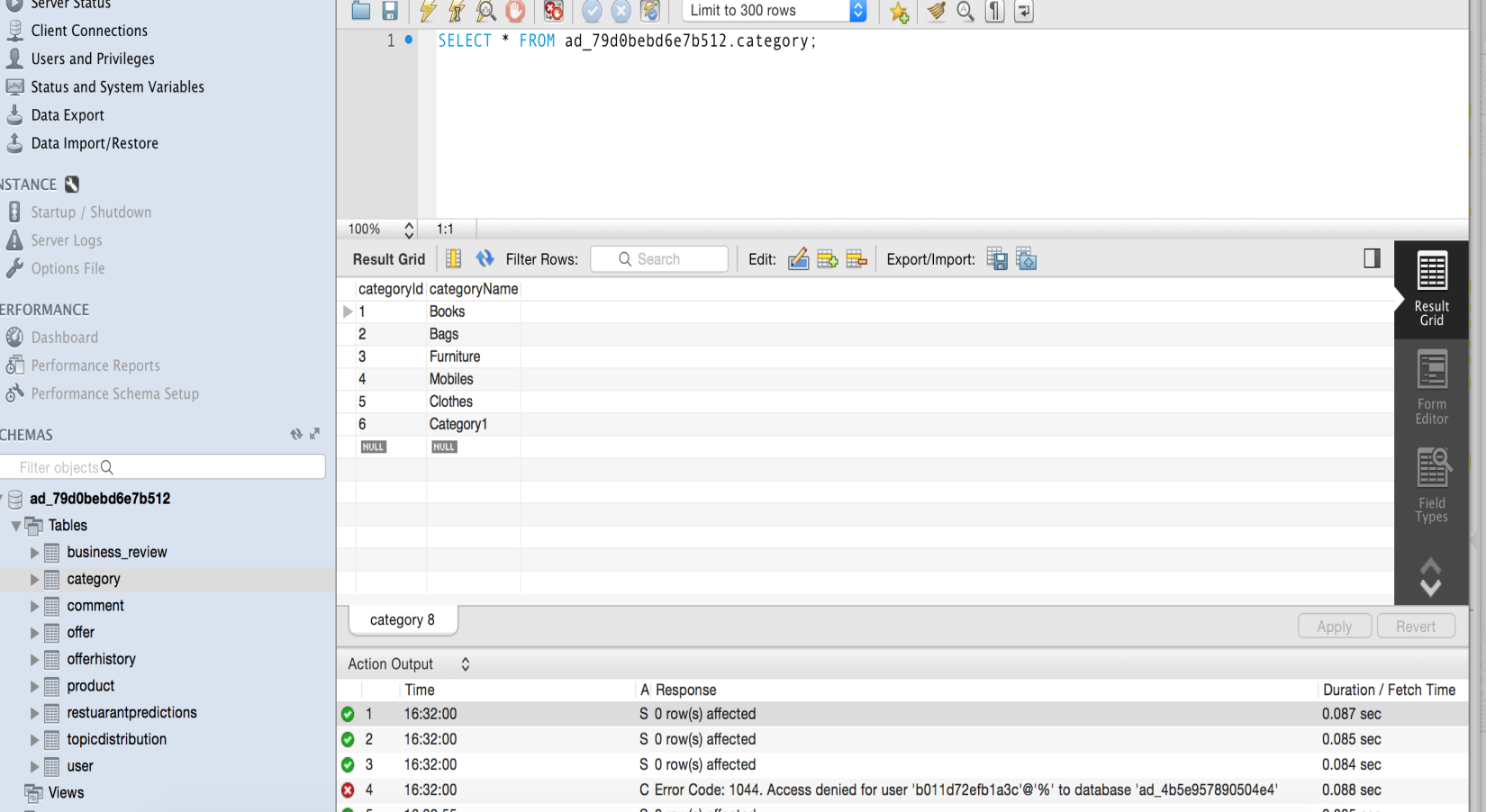
We have a shared MySQL DB instance on the cloud which all the servers can access. The purpose of this database is to manage the User, Category, Product and Offer data.

We have an instance in the ClearDb environment which is a secure, geo-distributed database-as-a-service. The purpose of having the database in the cloud is to ensure easy accessibility of the entire data to all the servers.

**Screenshots of database on Cloud:**







1. **Intra Cluster Load Balancer Server:**

One server in the cluster is the load balancer server. This server redirects every incoming request to one of the slave servers in a Round Robin fashion. The load balancer server does not process the request of perform any actions except for load balancing.

1. **Intra Cluster Slave Servers:**

Every server in the internal cluster except for the load balancer is considered as the slave server. These servers get the request from the load balancer and process it. They perform the required CRUD operations on the database and send the response back to the client.

1. **Inter Cluster Barter System**

Our individual barter system can also connect with other barter systems to create a Barter Community. Efficient inter cluster communication is ensured by all the barter systems mutually agreeing upon the format of the REST API calls.

**6. Database Schema**

All the teams mutually agreed upon a database schema that will handle all the requirements of a barter system.

The schema is as follows:

**Category:**

Category Information

CREATE TABLE IF NOT EXISTS `Category` (

`categoryId` INT(11) NOT NULL,

`categoryName` VARCHAR(45) NULL DEFAULT NULL,

PRIMARY KEY (`categoryId`));

**User:**

User Information

CREATE TABLE IF NOT EXISTS `User` (

`userId` INT(11) NOT NULL,

`firstName` VARCHAR(45) NULL DEFAULT NULL,

`lastName` VARCHAR(45) NULL DEFAULT NULL,

`emaiId` VARCHAR(45) NULL DEFAULT NULL,

`mobile` INT(11) NULL DEFAULT NULL,

PRIMARY KEY (`userId`));

**Product:**

Products posted by users with their information

CREATE TABLE IF NOT EXISTS `Product` (

`productId` INT(11) NOT NULL,

`productName` VARCHAR(45) NULL DEFAULT NULL,

`quantity` INT(11) NULL DEFAULT NULL,

`userId` INT(11) NULL DEFAULT NULL,

`expectedOffer` VARCHAR(100) NULL DEFAULT NULL,

`productDesc` VARCHAR(200) NULL DEFAULT NULL,

`productExpiryDate` DATE NULL DEFAULT NULL,

`isValid` BIT(1) NULL DEFAULT NULL,

`categoryId` INT(11) NULL DEFAULT NULL,

PRIMARY KEY (`productId`),

INDEX `userId\_fk1\_idx` (`userId` ASC),

INDEX `categoryId\_fk1\_idx` (`categoryId` ASC),

CONSTRAINT `categoryId\_fk1`

FOREIGN KEY (`categoryId`)

REFERENCES `Project2`.`Category` (`categoryId`)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT `userId\_fk1`

FOREIGN KEY (`userId`)

REFERENCES `Project2`.`User` (`userId`)

ON DELETE NO ACTION

ON UPDATE NO ACTION);

**Offer:**

Details of the offer made between two users

CREATE TABLE IF NOT EXISTS `Project2`.`Offer` (

`offerId` INT(11) NOT NULL,

`buyingQty` INT(11) NULL DEFAULT NULL,

`offeredDetails` VARCHAR(100) NULL DEFAULT NULL,

`buyerStatus` VARCHAR(45) NULL DEFAULT NULL,

`sellerStatus` VARCHAR(45) NULL DEFAULT NULL,

`offerExpiry` DATETIME NULL DEFAULT NULL,

`productId` INT(11) NOT NULL,

`buyerId` INT(11) NOT NULL,

`lastModified` DATETIME NULL DEFAULT NULL,

PRIMARY KEY (`offerId`),

INDEX `fk\_Offer\_Product1\_idx` (`productId` ASC),

INDEX `fk\_Offer\_User1\_idx` (`buyerId` ASC),

CONSTRAINT `fk\_Offer\_Product1`

FOREIGN KEY (`productId`)

REFERENCES `Project2`.`Product` (`productId`)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT `fk\_Offer\_User1`

FOREIGN KEY (`buyerId`)

REFERENCES `Project2`.`User` (`userId`)

ON DELETE NO ACTION

ON UPDATE NO ACTION);

**Comment:**

Comments to communicate in the offer

CREATE TABLE IF NOT EXISTS `Comment` (

`commentId` INT(11) NOT NULL,

`commentDesc` VARCHAR(45) NULL DEFAULT NULL,

`lastUpdated` DATETIME NULL DEFAULT NULL,

`offerId` INT(11) NOT NULL,

`userId` INT(11) NOT NULL,

PRIMARY KEY (`commentId`),

INDEX `fk\_Comment\_Offer1\_idx` (`offerId` ASC),

INDEX `fk\_Comment\_User1\_idx` (`userId` ASC),

CONSTRAINT `fk\_Comment\_Offer1`

FOREIGN KEY (`offerId`)

REFERENCES `Project2`.`Offer` (`offerId`)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT `fk\_Comment\_User1`

FOREIGN KEY (`userId`)

REFERENCES `Project2`.`User` (`userId`)

ON DELETE NO ACTION

ON UPDATE NO ACTION);

**OfferHistory:**

Track Offer for any change in quantity

CREATE TABLE IF NOT EXISTS `Project2`.`OfferHistory` (

`offerHistoryId` INT(11) NOT NULL,

`modified` VARCHAR(105) NULL DEFAULT NULL,

`lastModified` DATETIME NULL DEFAULT NULL,

`offerId` INT(11) NOT NULL,

PRIMARY KEY (`offerHistoryId`),

INDEX `fk\_OfferHistory\_Offer1\_idx` (`offerId` ASC),

CONSTRAINT `fk\_OfferHistory\_Offer1`

FOREIGN KEY (`offerId`)

REFERENCES `Project2`.`Offer` (`offerId`)

ON DELETE NO ACTION

ON UPDATE NO ACTION);

**7. Project Description and Screenshots**

1. **Load Balancing:**

We are using the http-proxy module in npm in order to implement load balancing.

http-proxy is a library that makes it easy to create proxies and load balancers using Node.js and JavaScript.

**Implementation:**

We have a server array containing the hostname and port information of all the servers connected in the cluster. We create a proxy server using http-proxy for each of the servers in the servers array. So we have an array of the proxy servers called proxies.

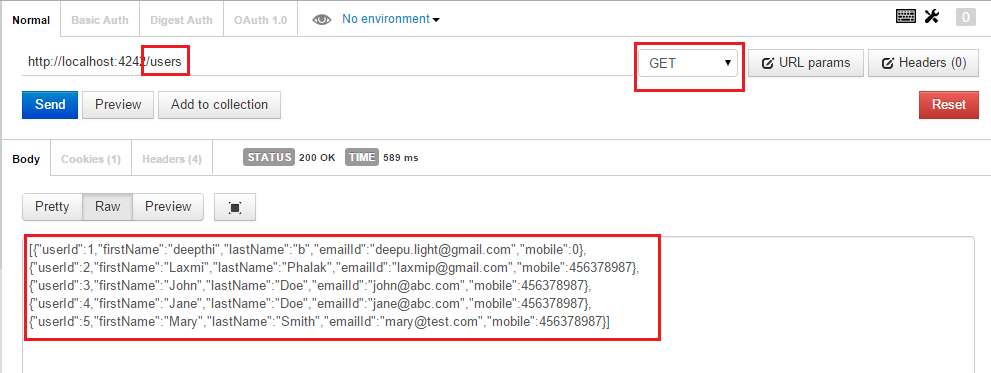
Every time there is an incoming request, the first proxy server in the proxies array is popped and is used to create an http server. Once the server is created and the request is processed, the proxy server is pushed back to the end of the proxies array creating a Round Robin management.

The http server makes sure that the ports of the servers in the internal cluster are never exposed i.e. the proxy server may be running on port 8080, but the http server is running on port 3000 and the user will only access the 3000 port without knowing about the servers running on 8080.

We have only one server acting as the load balancer. This server receives the request from the user and passes on every incoming request to the slave servers in a Round Robin fashion.

1. **REST API calls:**
2. **/users - GET**

Get the list of users in the system

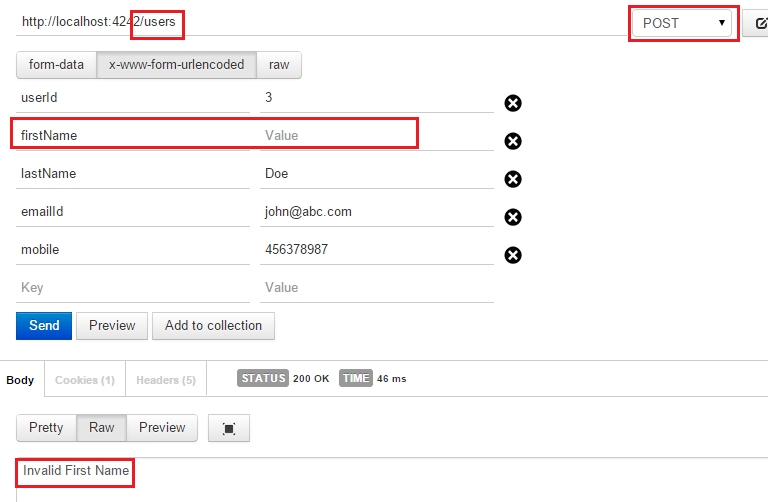


1. **/users - POST**

Create a new user

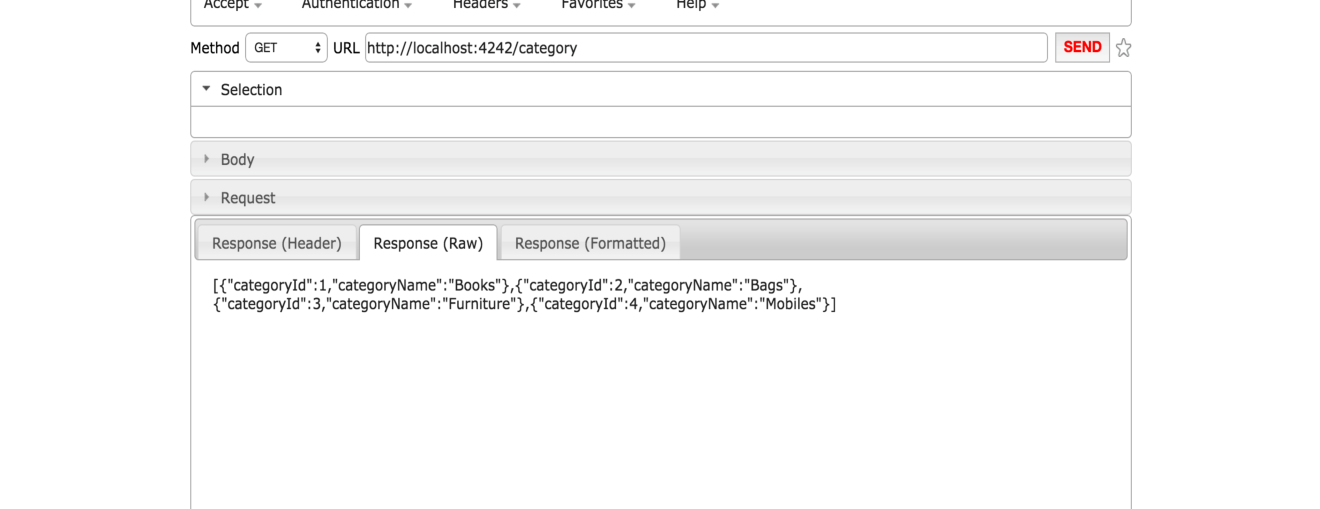


Error if any invalid input



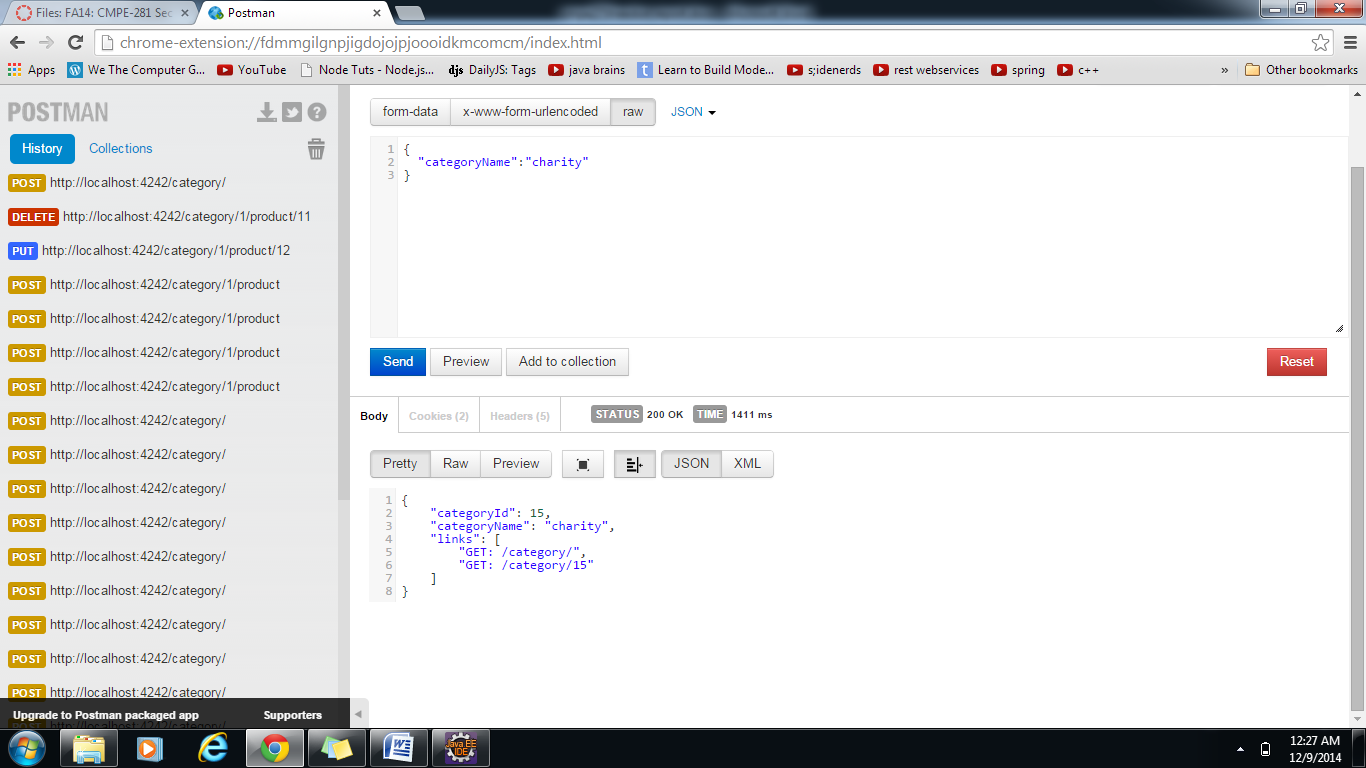
1. **/category - GET**

List of all categories in the system

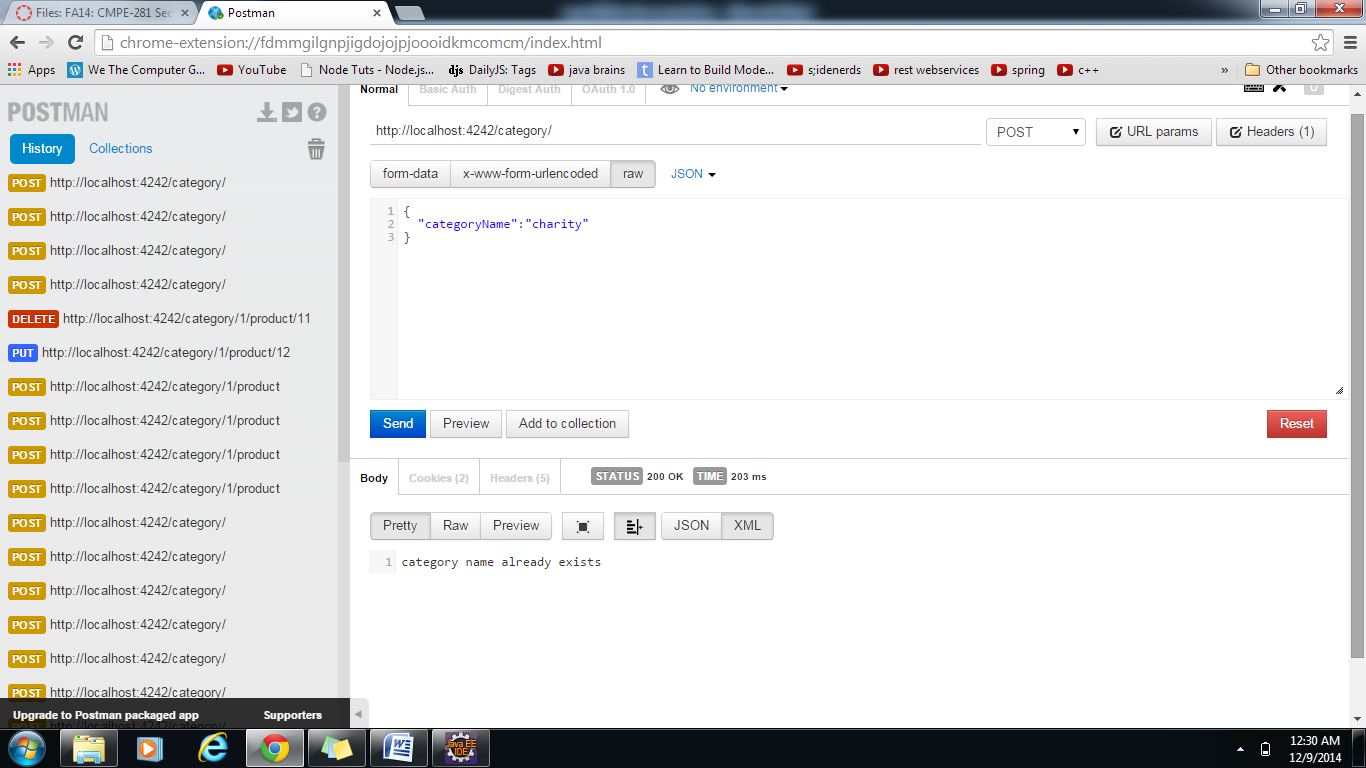


1. **/category - POST**

Helps user in creating a new category in JSON format as shown below. Validation of category name and its duplication is taken care. It also returns links with which user can proceed further.

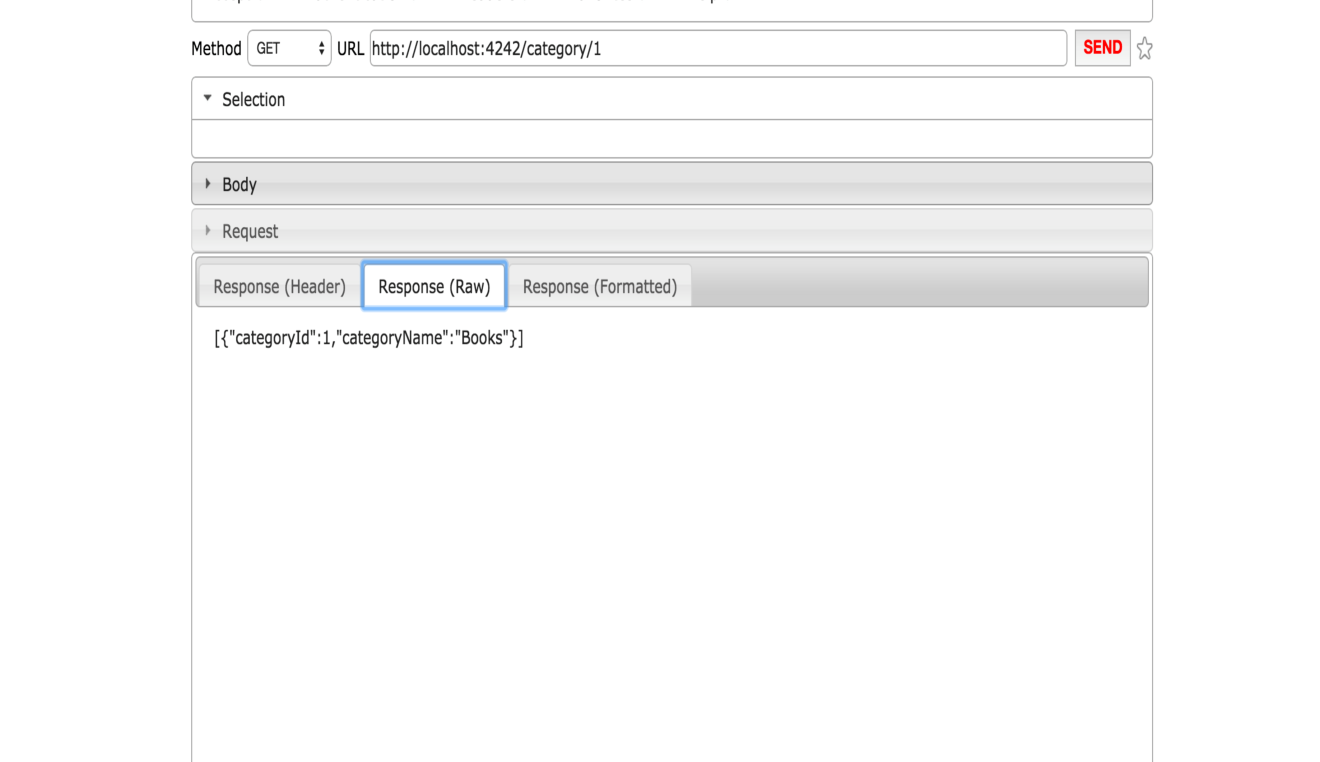


Validation for duplicate category names

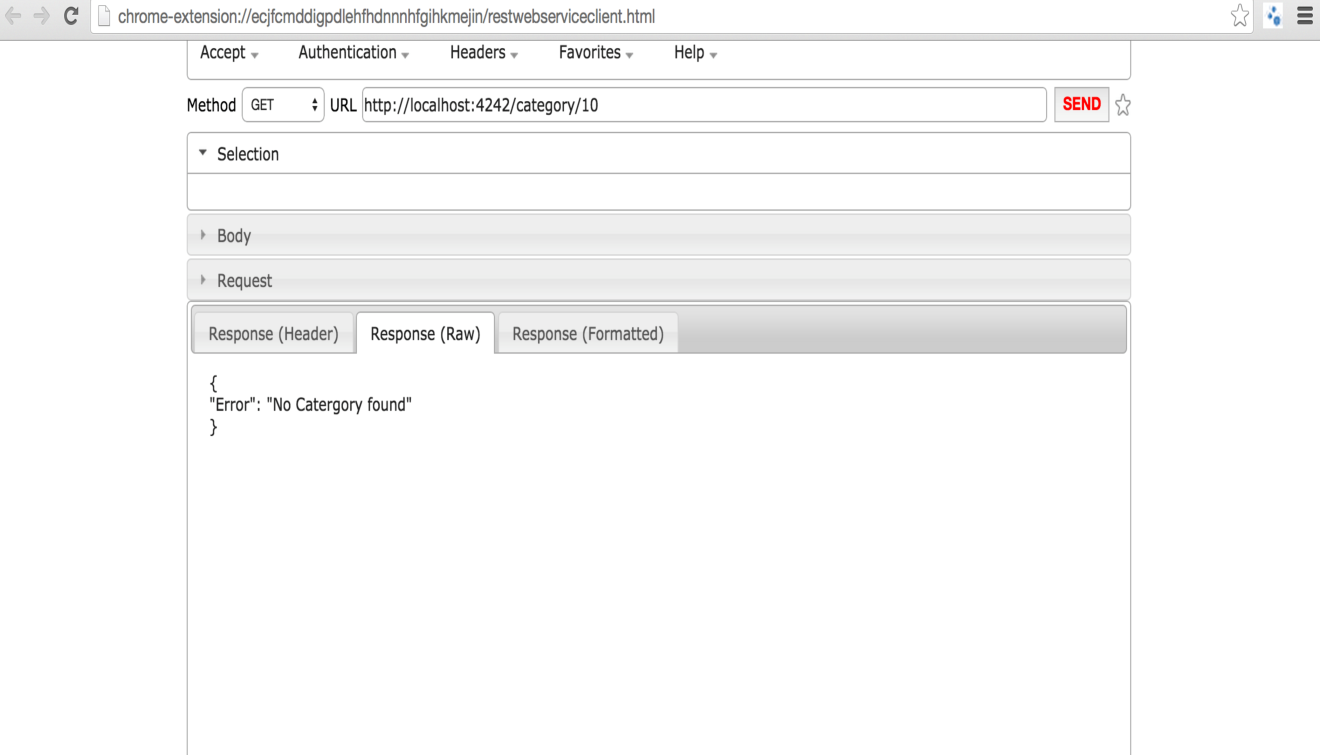


1. **/category/:categoryId - GET**

Get details of the category according to the category id

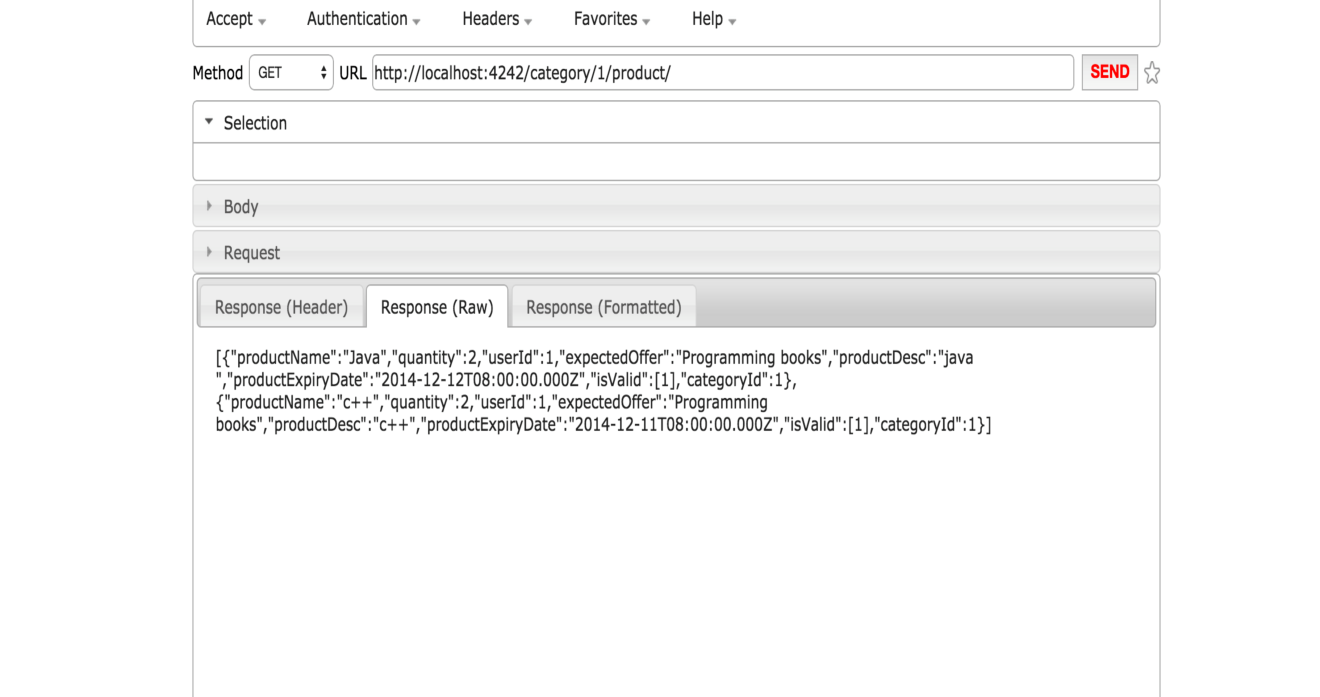


Error if Category not found



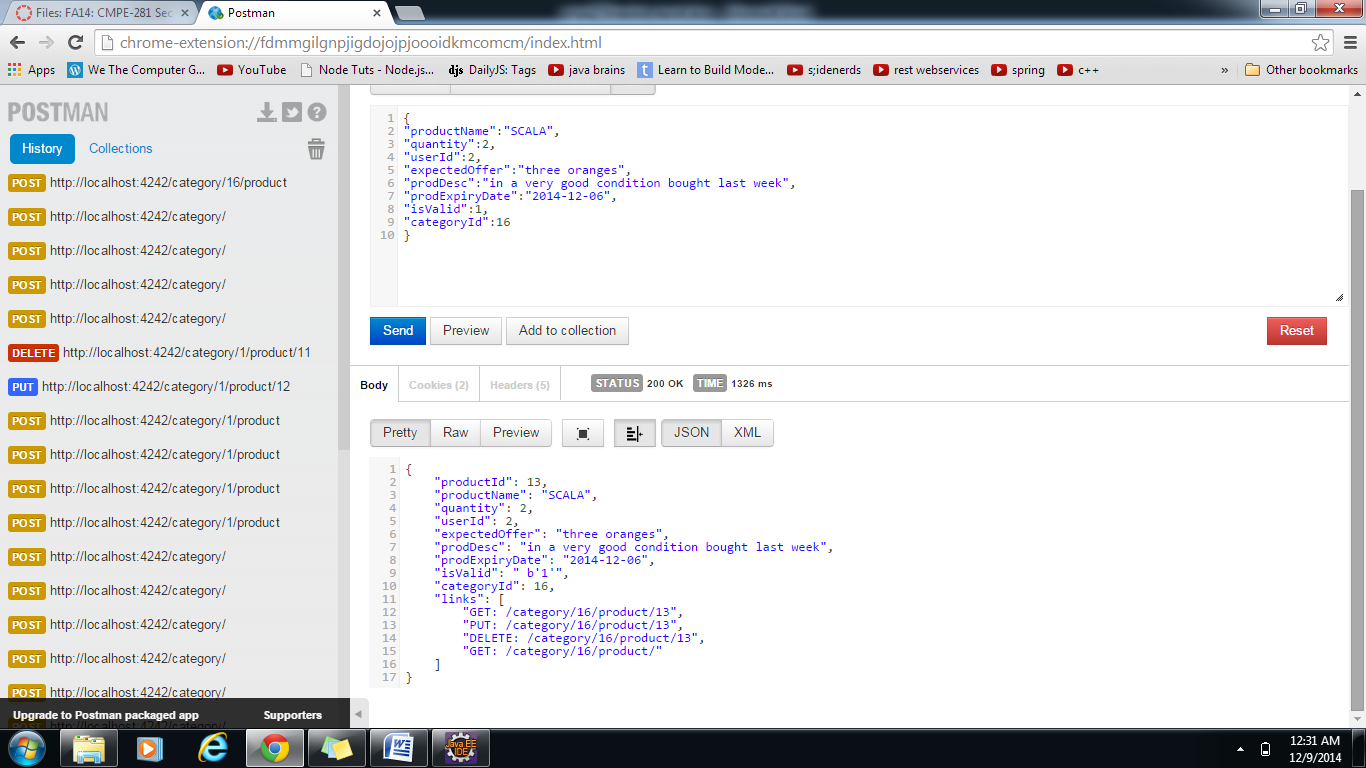
1. **/category/:categoryId/product/ - GET**

Get all the products under the category



1. **/category/:categoryId/product/ - POST**

The above url helps in creating a new product in a particular category. (:categoryId) is the req parameter.



1. **/category/:categoryID/product/:productId – PUT**

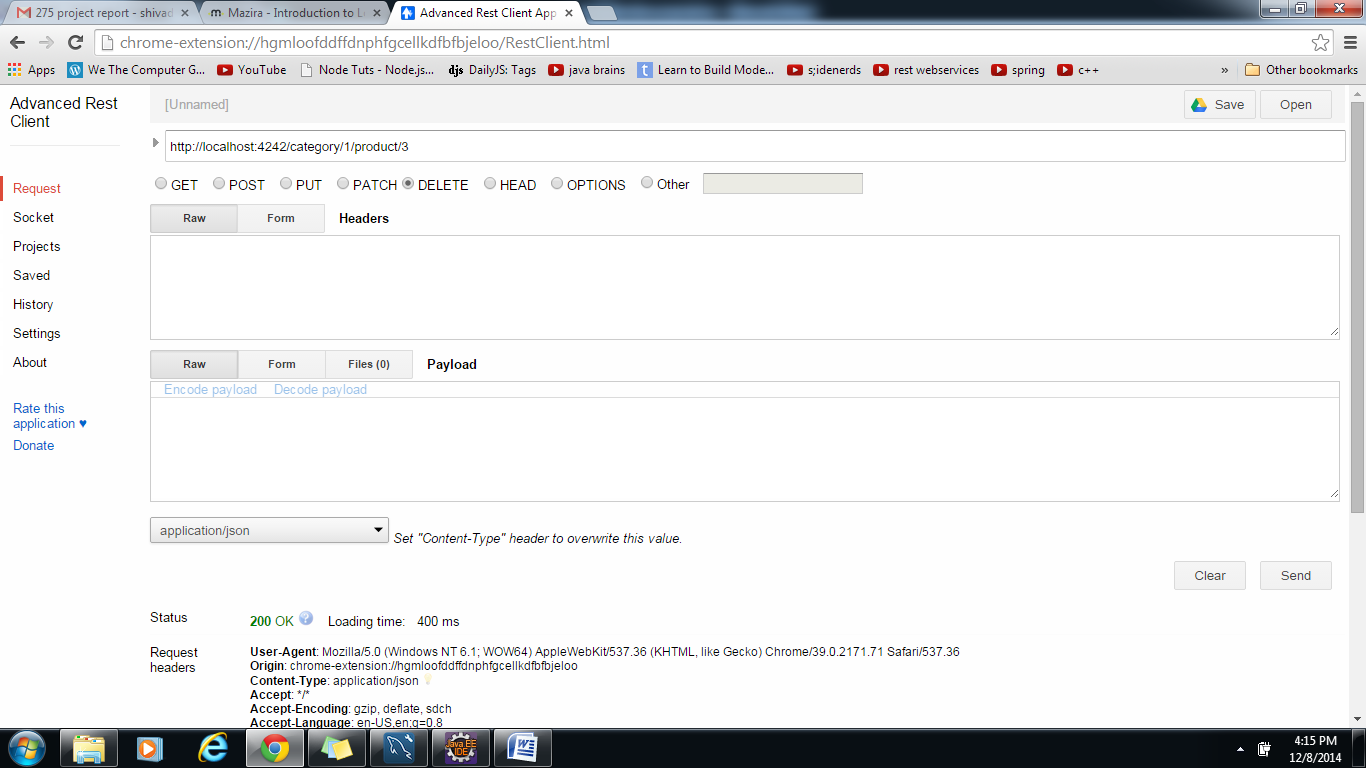
The above URL is helpful in updating product details.



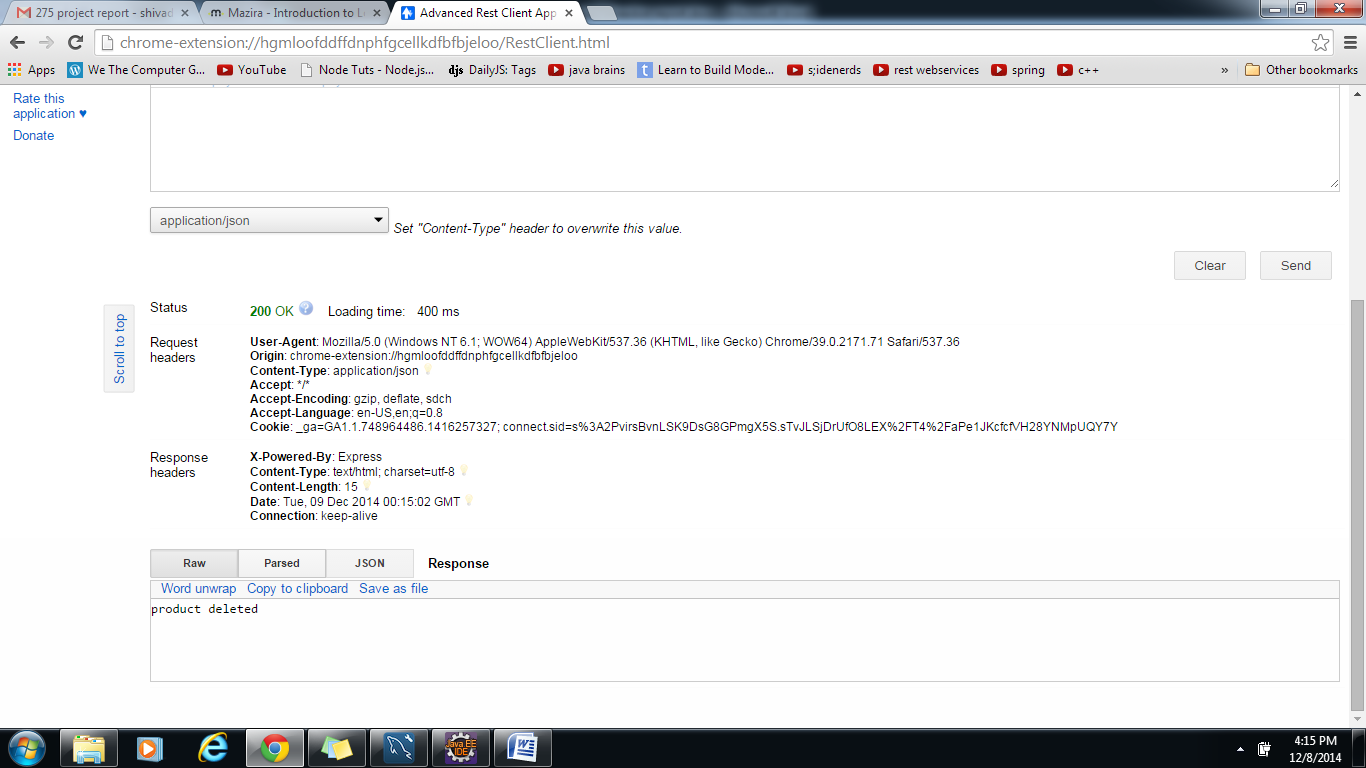
1. **/category/:categoryID/product/:productId – DELETE**

The above URL helps user in deleting a product.

REQUEST:



RESPONSE:



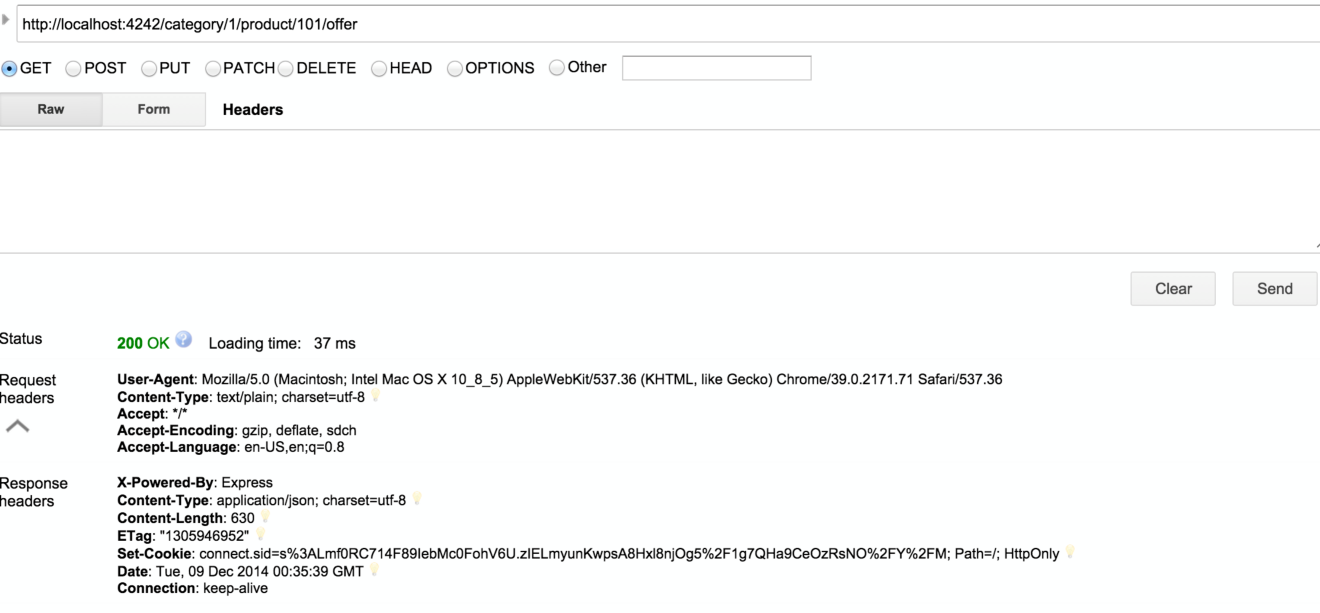
1. **/category/:categoryId/product/:productId/offer - GET**

Get list of offers made on particular product (optional pagination)

While posting a request for creating an offer in Offer table, we will provide the above mentioned fields. We are applying validations to make sure that the entries in the json format is in correct format. If any user who wants to buy a particular item(based on product ID) he/she can make a maximum of one offer on the same product. The buyer can negotiate with the seller by editing the details that he had provided and also via the Comments.

We have the check to make sure that the Id provided by the buyer in the json format is correct. Similarly the product id should also be present in the Product table. With the help of UI and UI validations we can ensure such validations.

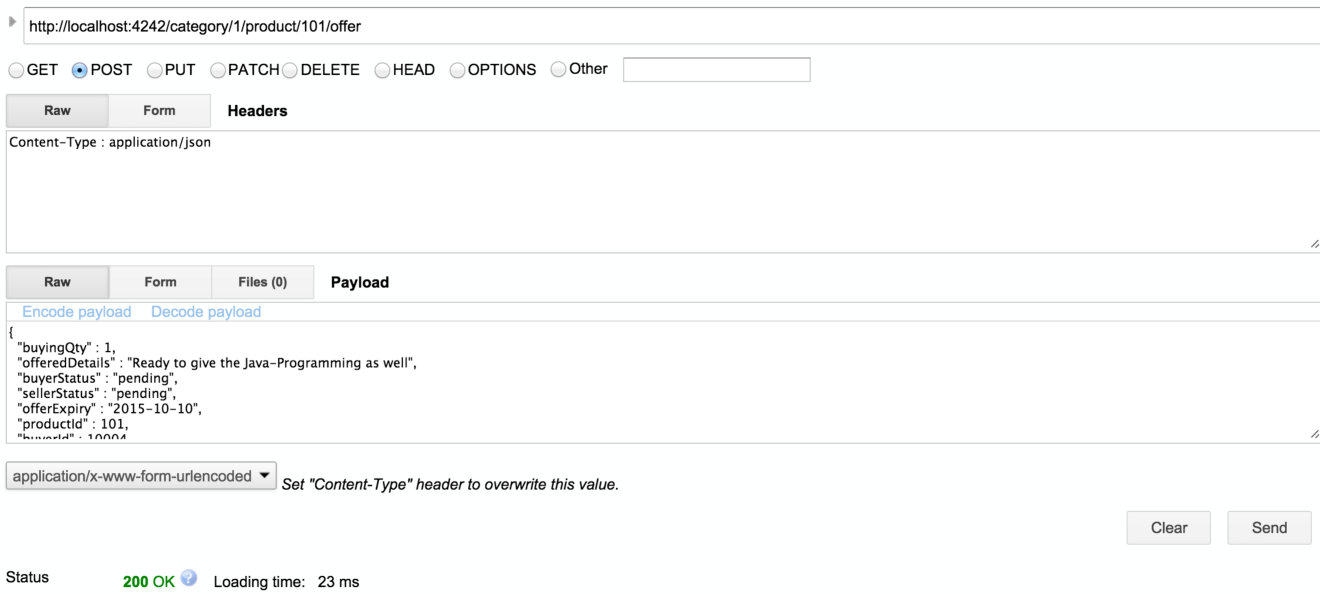
Given is the REST API for GET on Offer table, to display all the offers.

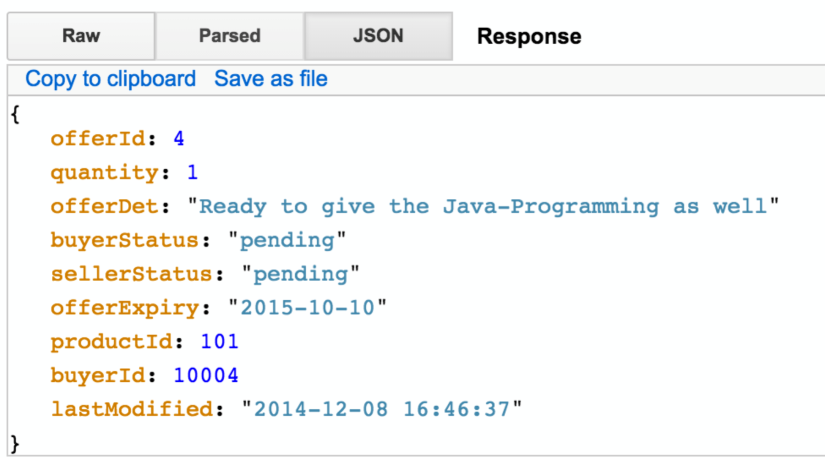




1. **/category/:categoryId/product/:productId/offer – POST**

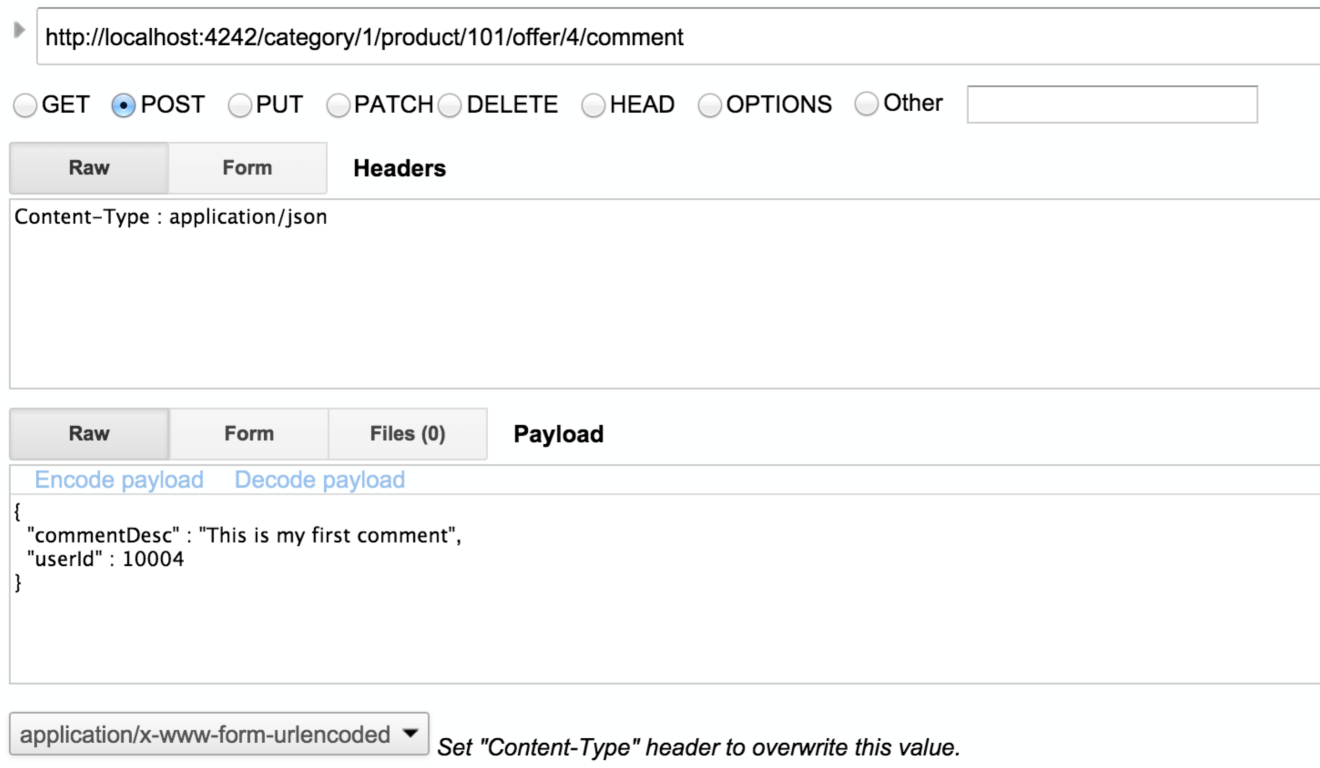
Post an Offer is shown below using the REST API call.

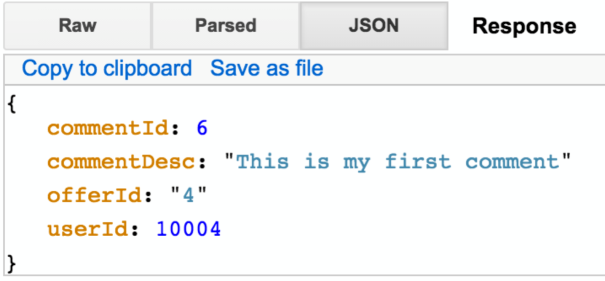




1. **/category/:categoryId/product/:productId/offer/:offerId/comment - POST**

Comments table keeps track of all the comments that has been happening between the buyer and the seller. We made sure that the data is properly populated. Also only buyer and a seller can comment upon a particular offer on any product. A third party can’t comment upon a specific offer.





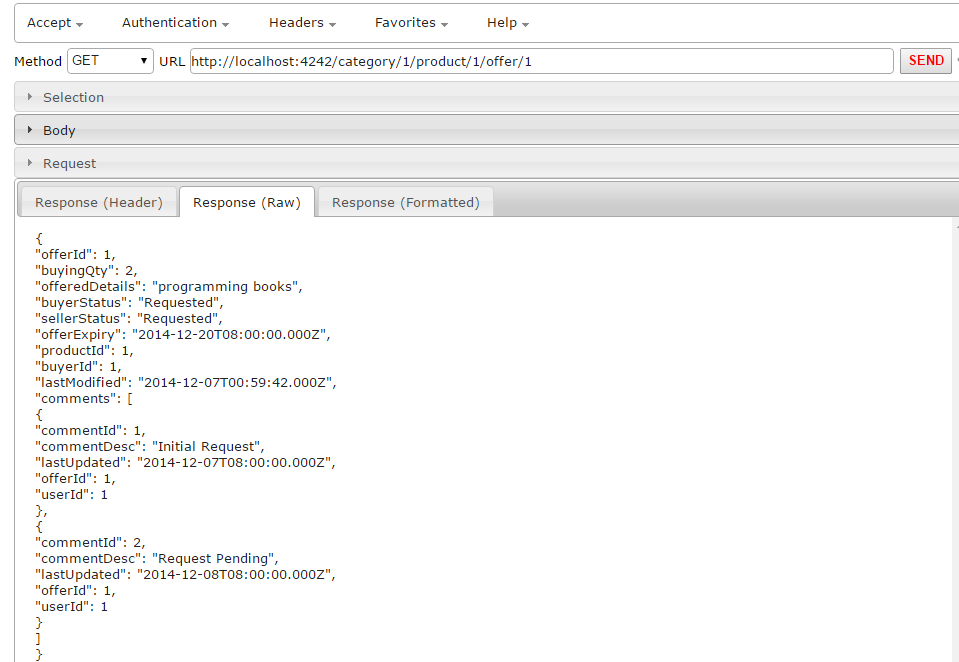
1. **/category/:categoryId/product/:productId/offer/:offered - GET**

This API uses the HTTP GET method to retrieve the details of a specific offer made for a product in a category. As an input, it accepts the offer id used to uniquely identify an offer among multiple offers made for given product id and category id, the product belongs to. This API retrieves the details of the offer made for the given offer id in the uri. As a buyer, the API enables to view the offer details made along with the updated comments. As a seller, the API enables to view the offer made by the buyer.

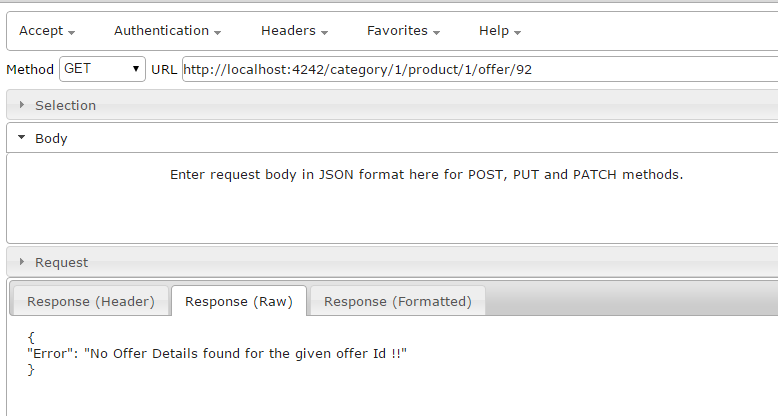
The system will fetch the buying quantity, offered details,buyer status, seller status, offer expiry, buyer id and last modified from the offer table based on the offer id and product id provided by the user. It will also fetch the list of comments for the offer id and generate a json response displaying all the details retrieved.

The system gives appropriate error messages if the offer id, product id or category id is invalid or if no data is available for the requested offer id.

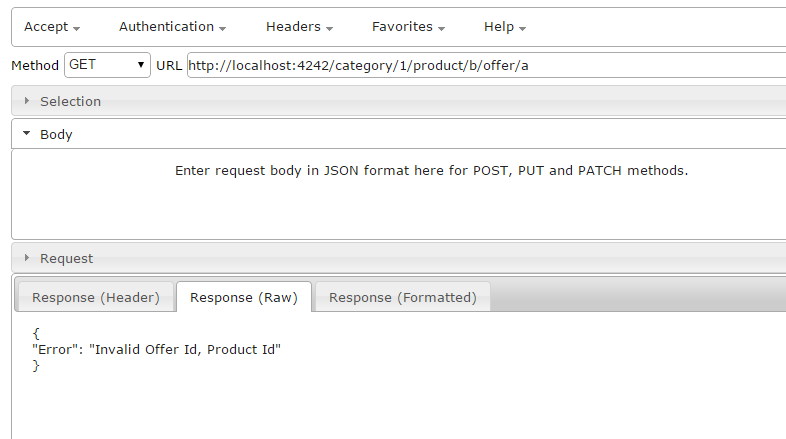
Valid Offer Id:



Offer Id not available:



Invalid offer id / category id/ product id:



1. **/category/:categoryId/product/:productId/offer/:offered - PUT**

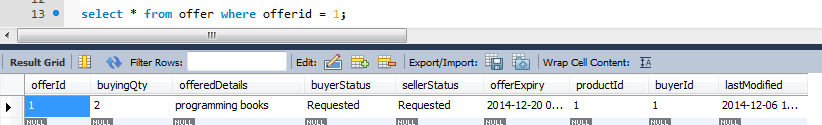
This API uses the HTTP PUT method to update the details of a specific offer made for a product in a category. The URI string contains the offer id used to uniquely identify an offer among multiple offers made for given product id and category id, the product belongs to. As an input, the API accepts the offer details that need to be updated for the specified offer id. It does not allow the user to update comments using this API.

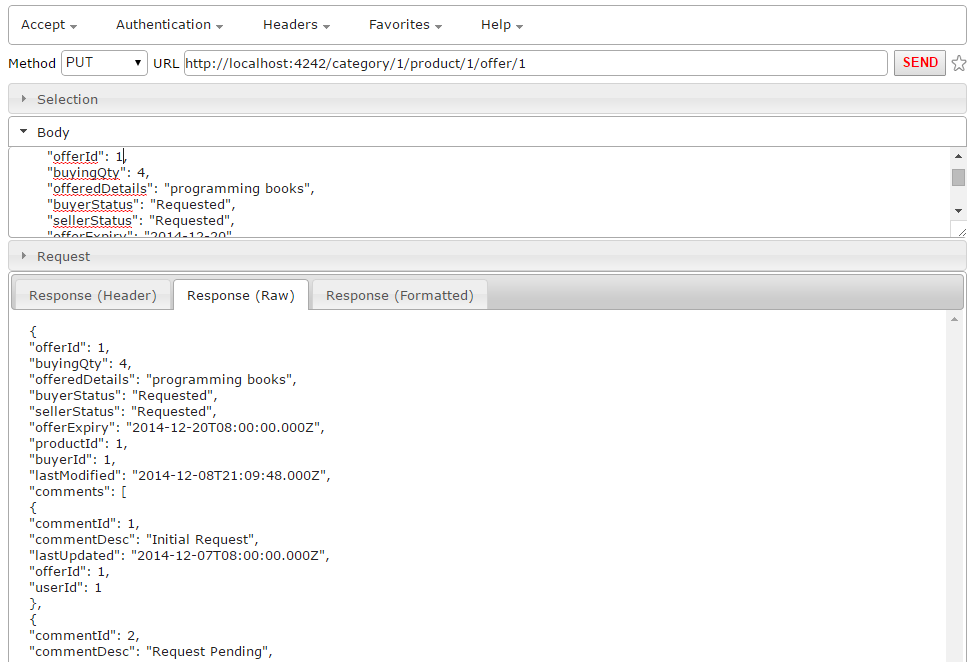
The buyer status and seller status can be 'Requested', 'Accepted' or 'Rejected'. Offer Expiry accepts date value. The last modified value is automatically updated when the offer details are updated in the database.

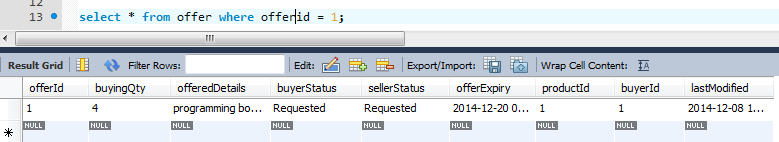
If the offer details are updated successfully, the system will send the response in json format.

If the update fails due to incorrect values in request parameters or incorrect offer id, the system responds with appropriate error message in the response with status 200.

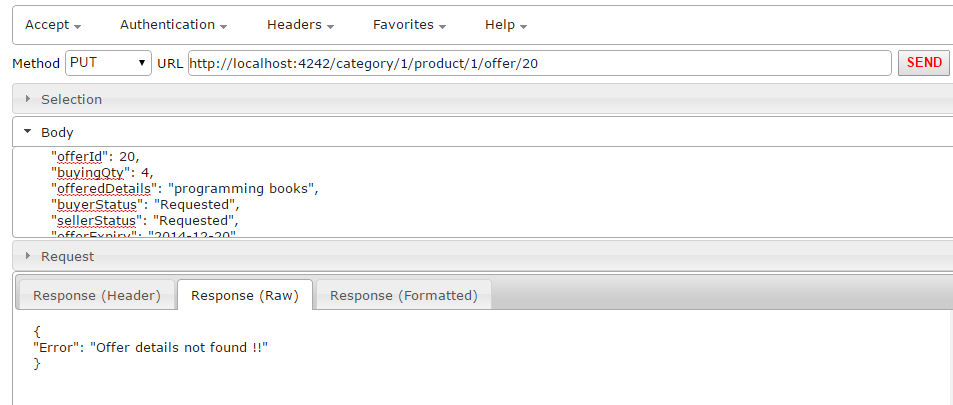
Update buying quantity:



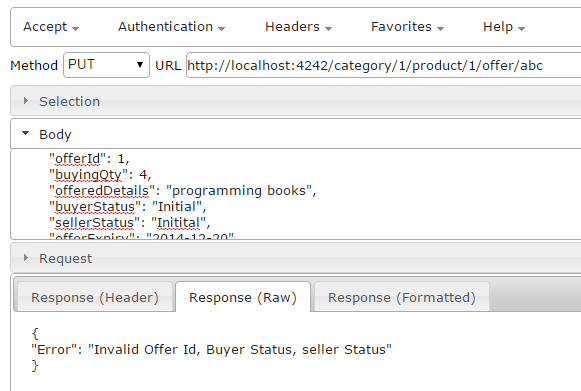




Offer id not available:



Invalid offer id/ product id/ category id and invalid status:

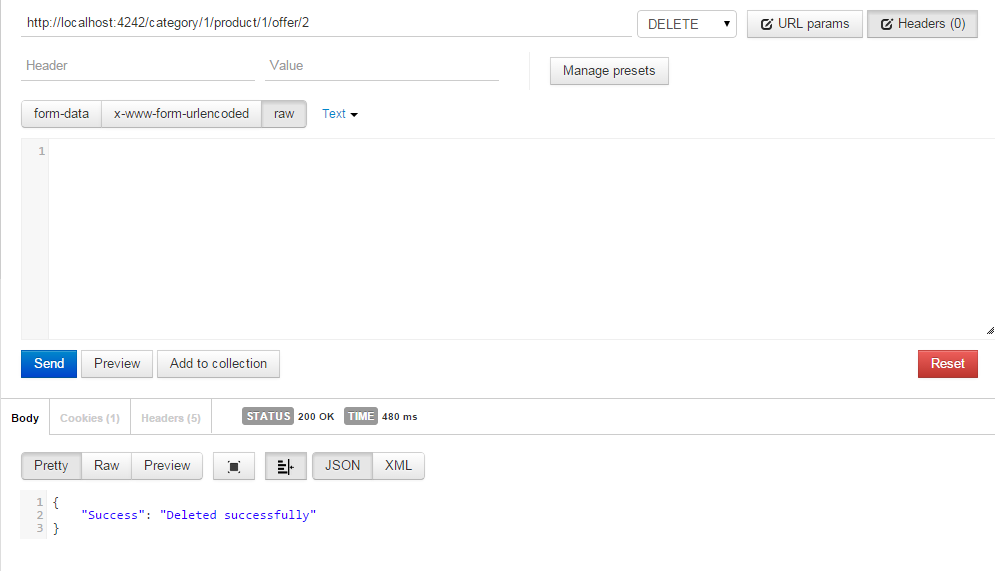


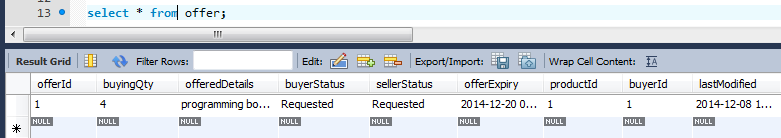
1. **/category/:categoryId/product/:productId/offer/:offered - DELETE**

This API uses the HTTP DELETE method to delete the details of an offer. The URI string contains the offer id used to uniquely identify an offer among multiple offers made for given product id and category id, the product belongs to. The system deletes the data related to this offer id from the offer table. It also deletes all the comments related to the offer id from the comments table.

If the delete is successful, the system sends a success message in response with status 200. If the delete is unsuccessful, it sends the error message along with status 400.





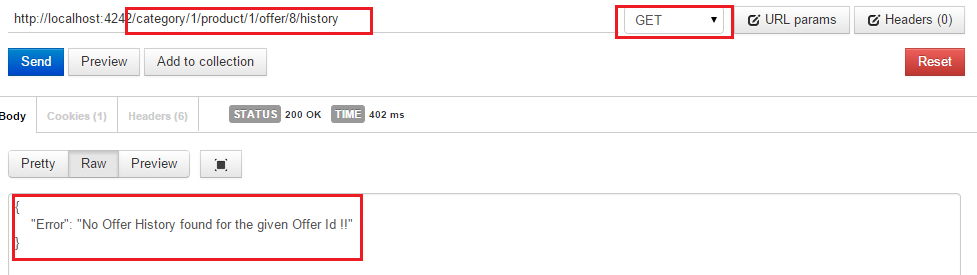


1. **/category/:categoryId/product/:productId/offer/:offerId/history – GET**

Gets the history of the offer i.e. tracks the changes made in the offer (quantity changed, etc.)



No Offer History found for Offer Id



1. **Testing:**

We used the SuperTest module in Node to test our system. SuperTest works by staring an express application and running tests against it. This type of testing is often referred to as end-to-end testing or sometimes as integration testing. This style of testing has the benefit of testing the public API where unit tests test individual internal units.

1. To begin add SuperTest to the package.json file under devDependencies

Package.json

"devDependencies": {  
"mocha": "\*",  
"should": ">= 0.0.1",  
"supertest": "0.3.x"  
}

1. Three modules should be added in node\_module folder, which is present inside the project folder. The modules are
   * mocha
   * should
   * supertest
2. Update your dependencies

npm install

1. SuperTest takes the express app that it will be testing as an argument; therefore we must export our app from app.js

app.js

var app = exports.app = express();

Example acceptance test:

In this we created a route that returns all the categories.

describe('GET /category', **function**(){

it('respond with json', **function**(done){

request(app)

.get('/category')

.set('Accept', 'application/json')

.expect(200)

.end(**function**(err, res){

**if** (err){ **return** done(err);}

done()

});

});

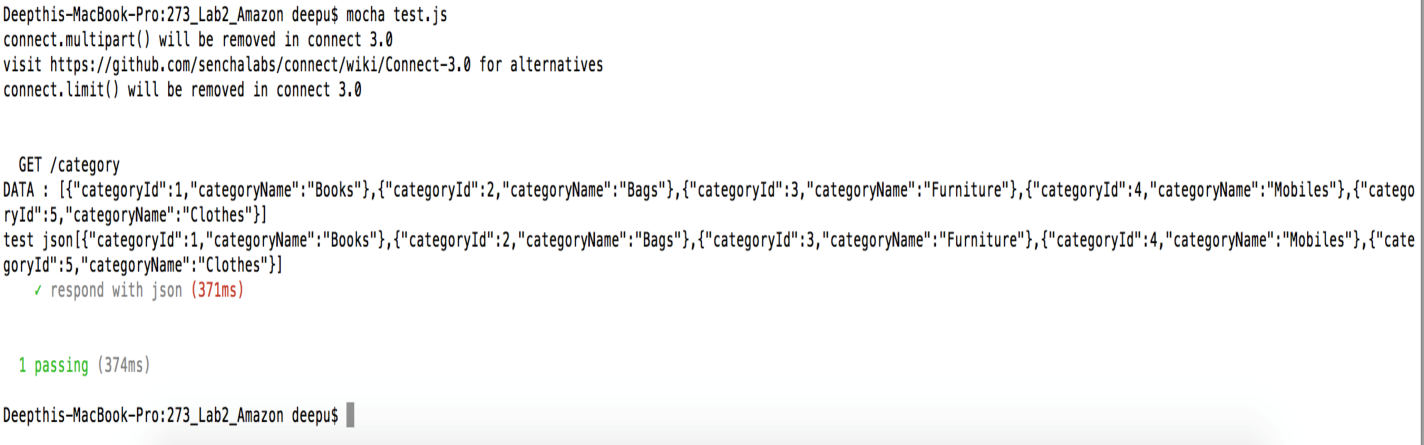
});

The route, which is used in the test case, is present in app.js.

To run the testcase navigate to the project folder from terminal and execute

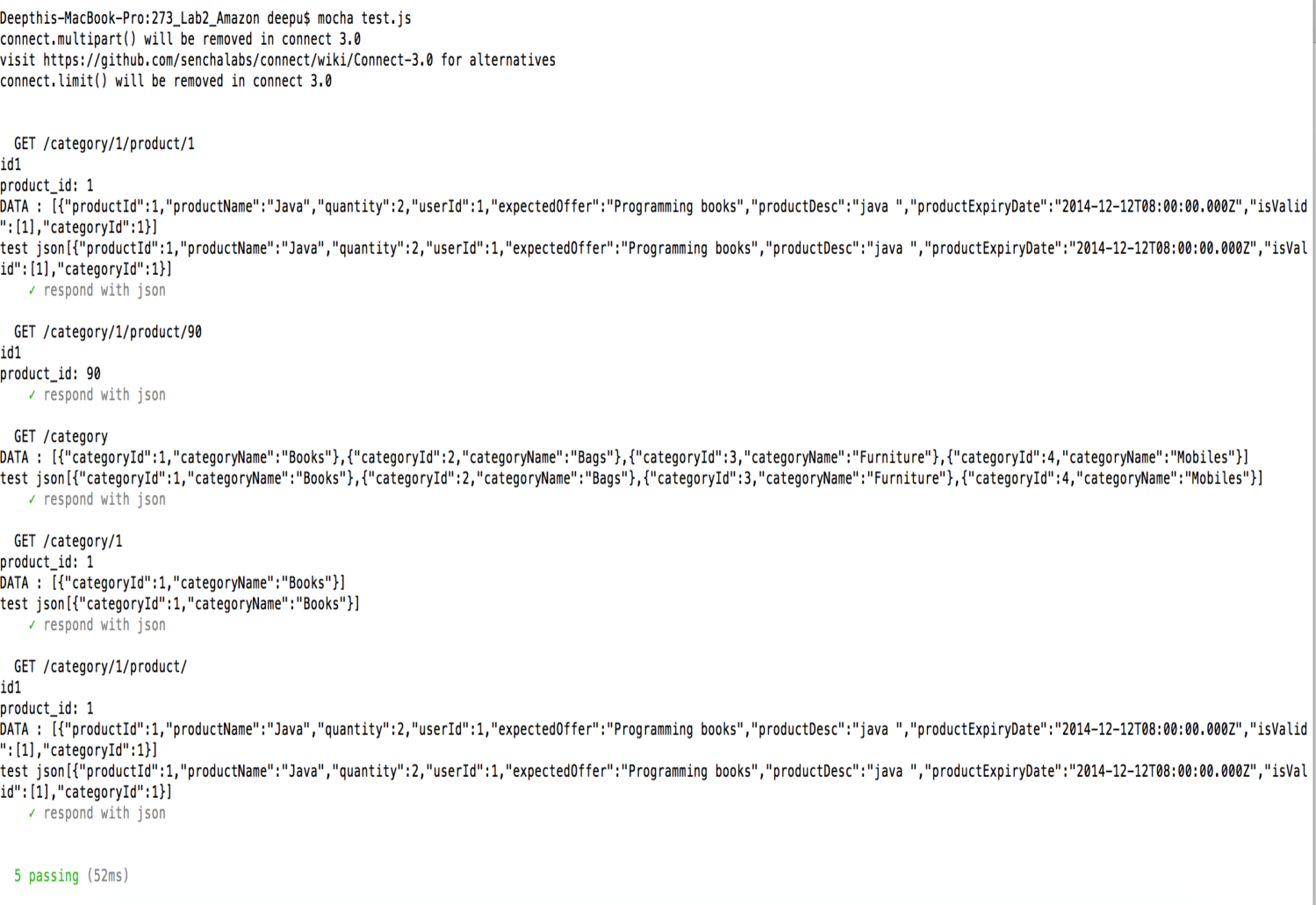
>mocha test.js

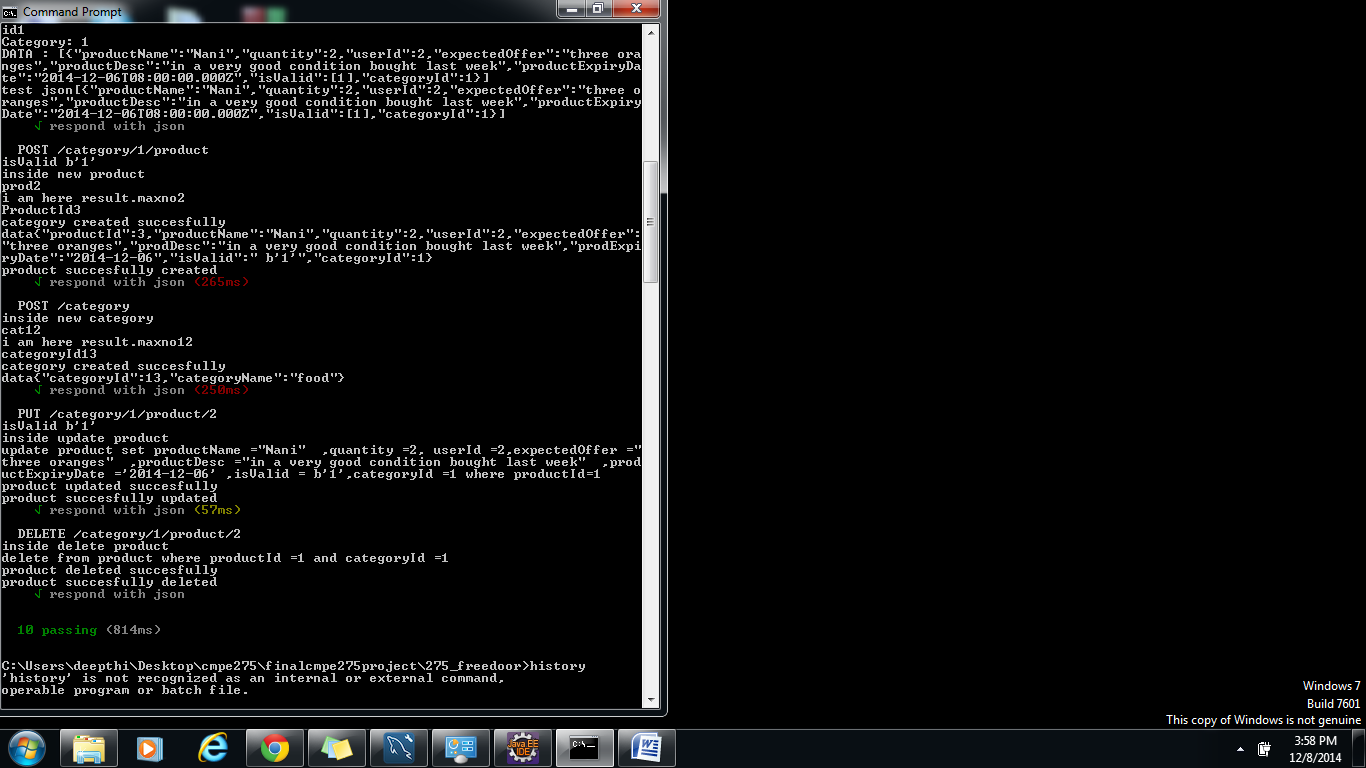
The output of the test case is as shown below.



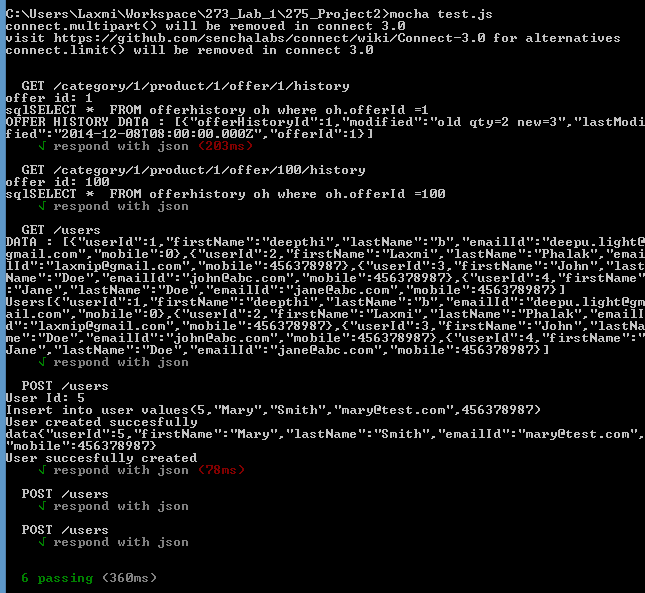
1. Similarly we created routes for the entire application.

**Screenshots for SuperTest on our system:**







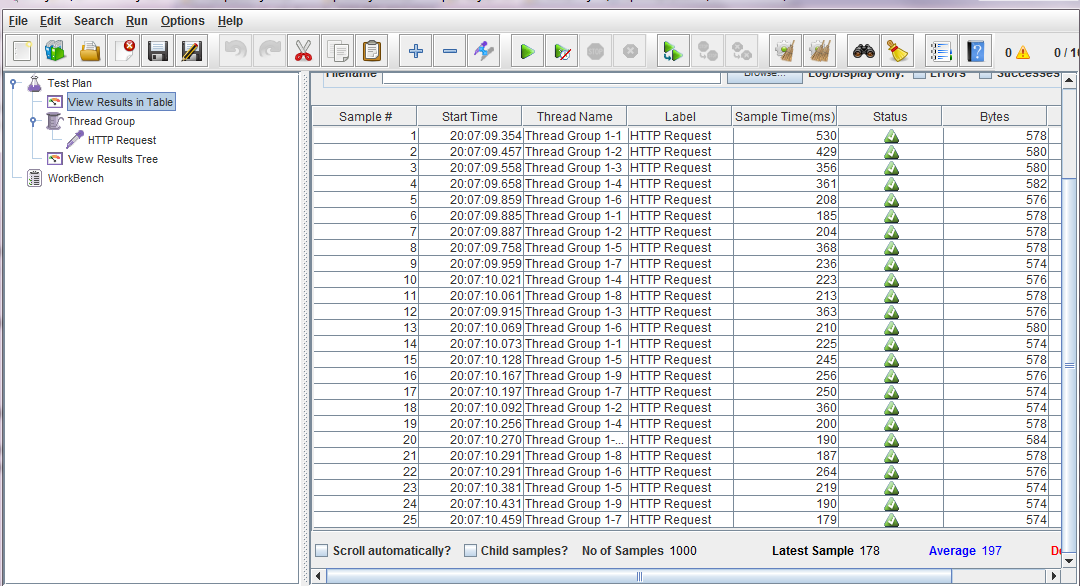


**JMeter load testing:**

We also tested our system for performance during under heavy load. We used JMeter to run 1000 requests hitting our system with /category GET call.

All the requests were successfully processed without the server getting crashed.

The output of the test is given below:



**8. Challenges**

Following are the major challenges we faced and consumed a lot of time during this project:

1. We wanted to implement the leader election for load balancing for failure tolerance. So we tried using the Seaport module in node. However the configurations turned out to be time consuming so we had to scrap the plan. Instead we used the http-proxy module for load balancing.