

San Jose State University, Spring 2013



Massive Online Open Course (MOOC)

CMPE 275 Project – 2

Submitted to Prof. John Gash

May 13, 2013

Submitted By:

Meet Mehta

Student Id: 008648939

Neel Anand

Student Id: 008664500

Rachana Belavigi

Student Id: 006400719

Snehal D'Mello

Student Id: 008647639

TABLE OF CONTENTS

1. Project Description
2. Technologies Used
 - 2.1 Python
 - 2.2 Django
 - 2.3 Bottle
 - 2.4 MongoDB
 - 2.5 PyMongo
3. Approach
4. Structure of API
5. Flow
6. MOOC testing using PyUnit
7. Things not used from Django
8. Project Status
9. Individual Contributions
10. References

Project Description

A massive open online course (MOOC) is an online course aiming at large-scale interactive participation and open access via the web. MOOCs provide interactive user forums that help build a community for the students, professors, and TAs. MOOCs are a recent development in distance education. ^[1]

The primary concept of this project is to develop a MOOC and connect it to various different MOOCs. Any user registered to a certain MOOC can access its own registered MOOC and also other teams' MOOCs through a defined Web Stack.

We have designed a MOOC in such a way that any user from different MOOC can easily access the functionalities of the MOOC which we created and can execute all the functionalities of the same.

The MOOC follows Django Web Framework for rendering the User Interface to the users. A certain defined User Interface is followed by every user who accesses our MOOC. The Bottle micro framework contains the back-end methods which are connected to the MongoDB database. The MOOC is totally developed under Python.

Technologies Used

Why Python?

Python emphasizes on code readability. It has readable syntax. It allows us to write fewer lines of code as compared to other languages. It's very fast to code and learn and follows advanced programming paradigms like Object-Oriented Programming, functional programming and imperative styles and also implements easy interoperability and cross-platform functioning.

Why Django?

Django is a web framework of python which gives lots of functionalities out of which the most important is the MVC architecture. The basic use of it is to create websites rapidly with reusability. It also provides the admin section to maintain website depending on the need (also need to configure it accordingly)

We have used templates from Django. It is similar to the master page concept in which base page remains the same and the child page extends that base page. It also allows us to put values at runtime in the declared block content (as we are using template, child must use `{{% block content %}}`). Thus, usage of Templates encourages a clean separation of application and presentation logic.

As shown in the figure, the following is the base HTML page which contains the template variable which are to be changed according to the child page. The child page would come under the `{% block content %} {% endblock %}`.

```
{% load staticfiles %}
```

- [Home](#)
- [Categories](#)
- [Courses](#)
- [Discussions](#)
- [Announcements](#)
- [My enrolled Classes](#)
- [Account Settings](#)
- [Logout](#)

CmpE 275 MOOC

Team Jargons

```
{% block content %}{% endblock %}
```

© Copyright Team Jargons

The child page is given below. The Curly braces block will render the base HTML page.

```
{% extends "base.html" %} {% block title %}Login{% endblock %} {% block content %} {{name}}
```

[Login](#)

User Name :

Password :

Not Registered yet? [Click Here](#) to Register.

```
{% endblock %}
```

Adding to that, we encountered some problem linking CSS and JavaScript files onto the templates. After researching, we came upon the solution of implementing *staticfiles* concept.

This functionality separates the content i.e. static, which would remain static throughout the website. The files like CSS files and JavaScript files in the *static* folder, serves additional functionality towards rendering the web content while in the use of templates. Django has this functionality of linking CSS, Javascript files or image files through this concept of static files.

For that, we need to define a URL for the files we need to define static; the path would come under `STATIC_URL` parameter in the *settings.py* file.

On whatever page we want to add those files, we would add the following content:

```
{% load staticfiles %}  

```

Here, the `{% load staticfiles %}` would render the path accordingly and fetch the desired path of the static file.

Again, the best thing of Django is that you can use it in any IP/Port with single command “python manage.py runserver 192.168.0.1:8080” at the time of starting the server which is not easy to implement in case of java running Apache Tomcat.

Why Bottle?

Well, Bottle is also a framework similar to Django for python but it's a micro-framework. It consists of a single bottle.py file to run the whole framework which again supports both python versions v2.x and v3.x.

Bottle offers request dispatching with URL parameter support, templates, a built-in HTTP server and adapters for many third party WSGI/HTTP server and template engines.

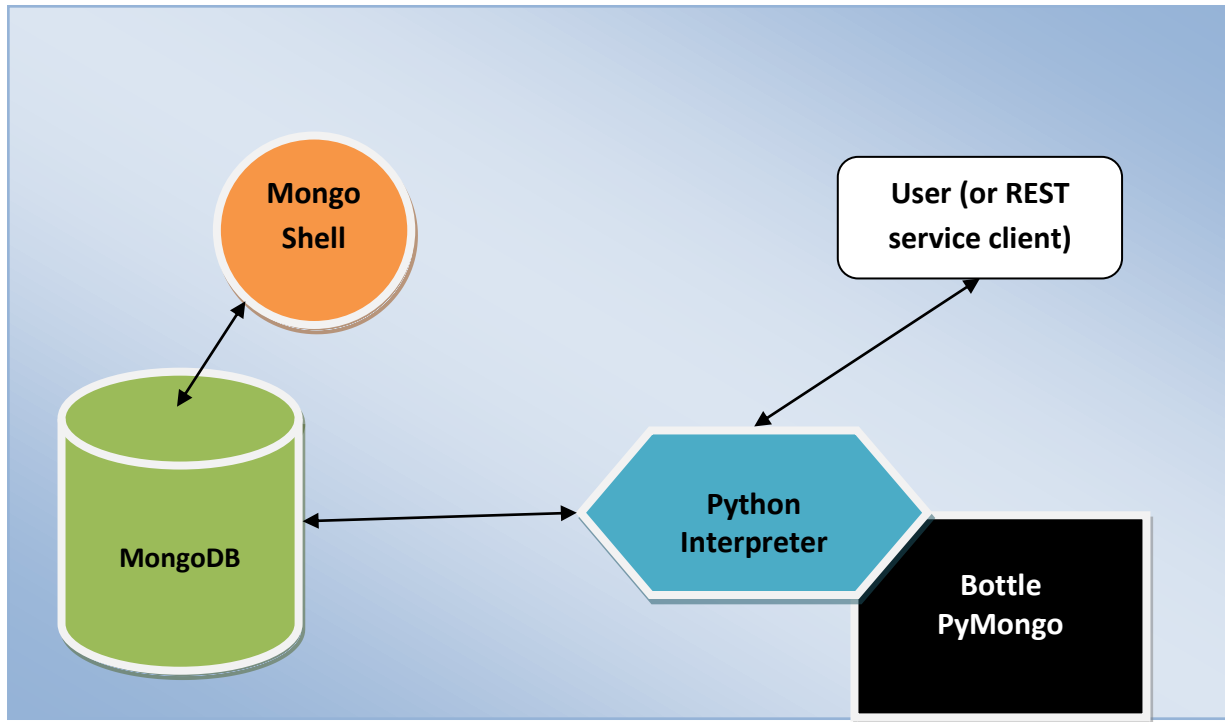
Why MongoDB?

MongoDB is an agile and scalable open-source NoSQL database. It supports Document-oriented storage. It means that it stores the structured data as JSON-like documents making the execution and integration of the data in the application faster and easier.

Here, as we are using JSON for serializing and transmitting the data, we are more inclined towards using MongoDB as it follows JSON-style documents. So, it would be easier to transmit and fetch the data from the database and vice-versa.

Why PyMongo?

PyMongo is a Python driver for MongoDB. It's a distribution made in Python which contains tools that works with MongoDB. As we are using Python as a primary language for the MOOC and MongoDB as a backend database, we tend to use PyMongo to connect.



Web Server and RESTful APIs:

Bottle is a fast, simple and lightweight WSGI(Web Server Gateway Interface) micro web-framework for Python. The Web Server Gateway Interface(WSGI) defines a simple and universal interface between web servers and web applications or frameworks for the Python programming language[1]

We decided to use Bottle framework as it is very lightweight and does not depend on any external libraries. We can start using Bottle by just downloading bottle.py in our project directory. Writing a simple web service using bottle is as simple as :

```
from bottle import route, run, template

@route('/hello/:name')

def index(name='World') :

    return template('<b>Hello {{name}}</b>!', name=name)

run(host='localhost', port=8080)
```

Representational State Transfer (REST) is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged as a predominant web API design model[2]. The general concept of a web API (application programming interface) has two main interpretations. It is used to refer to both a server-side API upon a web server as well as client-side API within a web browser. A server-side web API is a programmatic interface to a defined request-response message system, typically expressed in JSON or XML, which is exposed via the web—most commonly by means of an HTTP-based web server[3]

The REST architecture was implemented in two sections. The first was the request handler which handled all the incoming requests and second were the resources. The request handler(moo.py) handled all RESTful APIs using “@route()”. The request handler was responsible for first level of validation, handling exceptions and returning appropriate JSON response.

All resources were implemented as python classes. The resources supported were: *user, course, category, quiz, announcement, discussion, messages and admin*. Each resource was responsible

for establishing connection to MongoDB , perform second level of validation and querying MongoDB based on the incoming request. All the core computations were handled within each resource.

We followed the naming conventions as given in the common document that was shared throughout the class. However, we found that proper RESTful API standards were not followed. As part of our implementation, we supported both the “should be” RESTful APIs and APIs as suggested in the common document. This was done to maintain uniformity of APIs throughout the class. Our RESTful APIs were based on standards provided in:

<http://www.restapitutorial.com/lessons/restfulresourcenaming.html>

<http://www.restapitutorial.com/lessons/restquicktips.html>

Suggested URL : <http://localhost:8080/course/enroll>

Corrected URL : <http://localhost:8080/user/:email/course/:id>

The Suggested URL, requires the user email and course id to be provided as part of the post data. However, course enrollment action is always associated with the user, and should be a sub-resource under the user resource; and hence the Corrected url.

Also, the common shared document didn't mention the input requests and the output responses and their format. To resolve this, we re-documented the APIs with the missing information. The updated API documentation is as follows:

MongoDB:

MongoDB is a schemaless database. In MongoDB, when a record is inserted into the database, an unique id is generated and returned. This id is stored in database as “_id” and is stored in a ObjectId format. Thus, when we retrieve data from the database, it is retrieved as an Object. So, when we try to convert it into json, it throws error as “Object is not serializable”. Thus in order to serialize the object, we need to convert it to string. So, I implemented a custom JSON encoder which takes care of serialization errors:

```
class MongoEncoder (JSONEncoder) :
```

```

def default(self, obj, **kwargs):
    if isinstance(obj, ObjectId):
        return str(obj)
    else:
        return JSONEncoder.default(obj, **kwargs)

def encodeResponse(data):
    return json.dumps(data, cls=MongoEncoder)

```

Approach

We made separate tables (Classroom/Sites) for maintaining different IP/Ports. You can add IP/Port via dedicated Admin interface (username=meet & password=meet) provided in Django. You also have the option to select default_url by which your backend always connects to specific service of running of that IP and port (in our case its bottle).

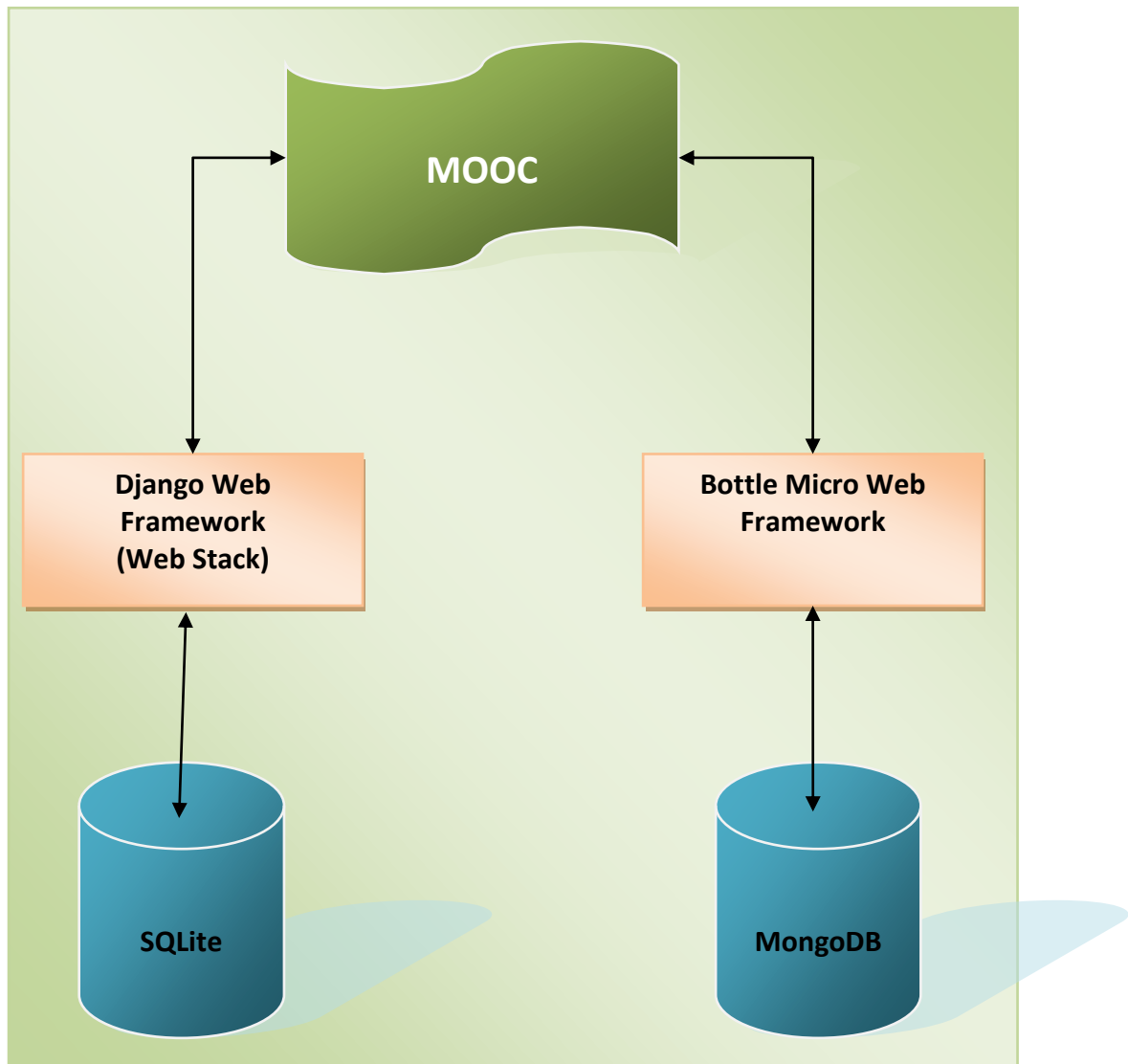
Note: While selecting default_url (in classroom/sites table) do not forget to uncheck previous default_url else you will tend to get errors.

Moreover, we made the connections to the service decouple to default_url. i.e. you can change your default_url from admin interface and your backend will automatically connect to that service.

Initially, we are unaware of how to access the data from Django, so we directly made connections to db and fetched the data. But then we came to know that we can also fetch the data from models directly. So we adapted that concept. We made show_url() method inside classroom/views.py file to directly access the data using models instead of making the connections to database manually. You can also see connections() method inside classroom/categories.py file which we are directly making connections to SQLite3 for fetching the data. We know that we are wrong and making the connections manually instead of Django, we make our classroom/users.py to use it in a Django way (fetching the data from models)

We are using Django's inbuilt method for making authentication of user, login and logout. We again find one bug (well not really a bug) while doing a authentication and login using Django

functionality takes more time as comparatively (May be a cold start problem but still not sure, did not get enough time to check measure precise time).



Structure of API

/user

- 1) create User (Register)
- 2) sign In
- 3) sign Out
- 4) update
- 5) delete
- 6) enroll

/category

- 1) add
- 2) list

/course

- 1) add
- 2) edit (also includes add & show announcement related to that specific course)
- 3) list

/course/announcement

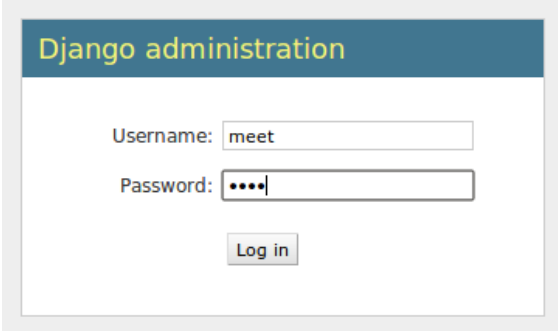
- 1) add --> only the person adding course can add announcement.
- 2) edit --> only the person adding course can add announcement.
- 3) list

/course/discussion

- 1) add
- 2) list

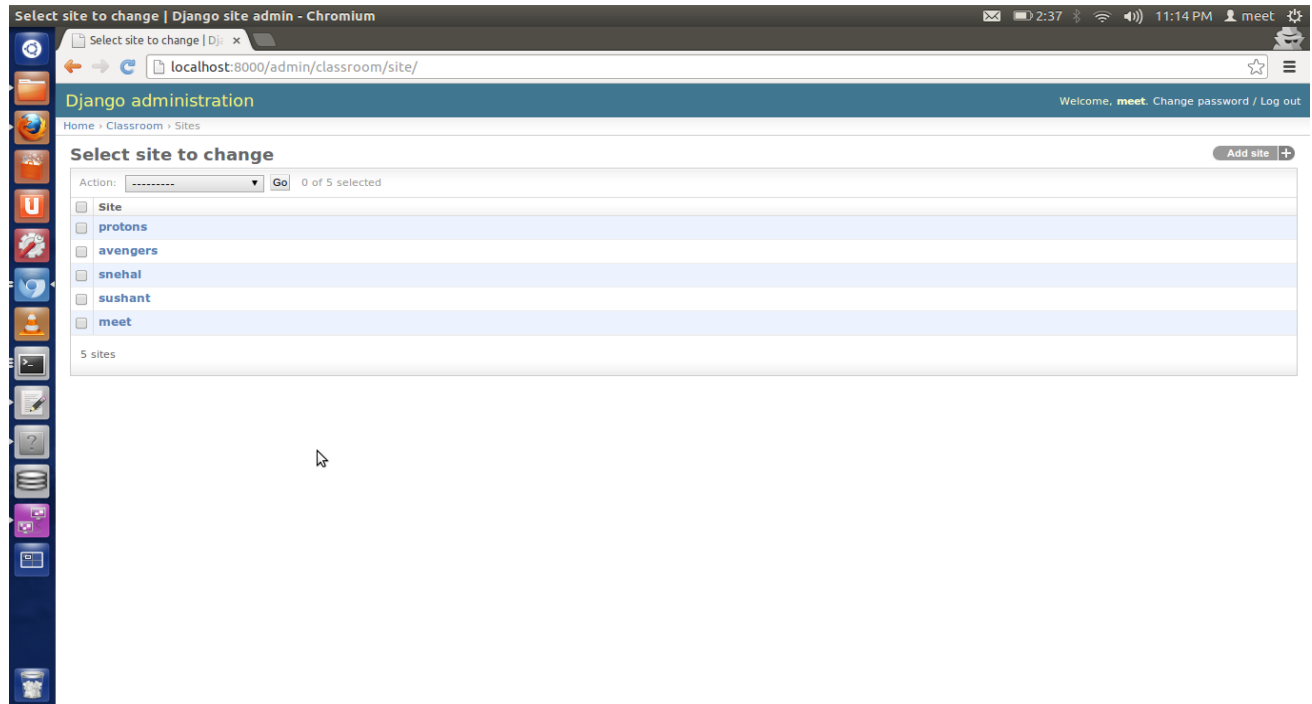
Flow:

1. Go to Django server url (host:port/admin/) (login with username-meet and password-meet) as shown in below figure

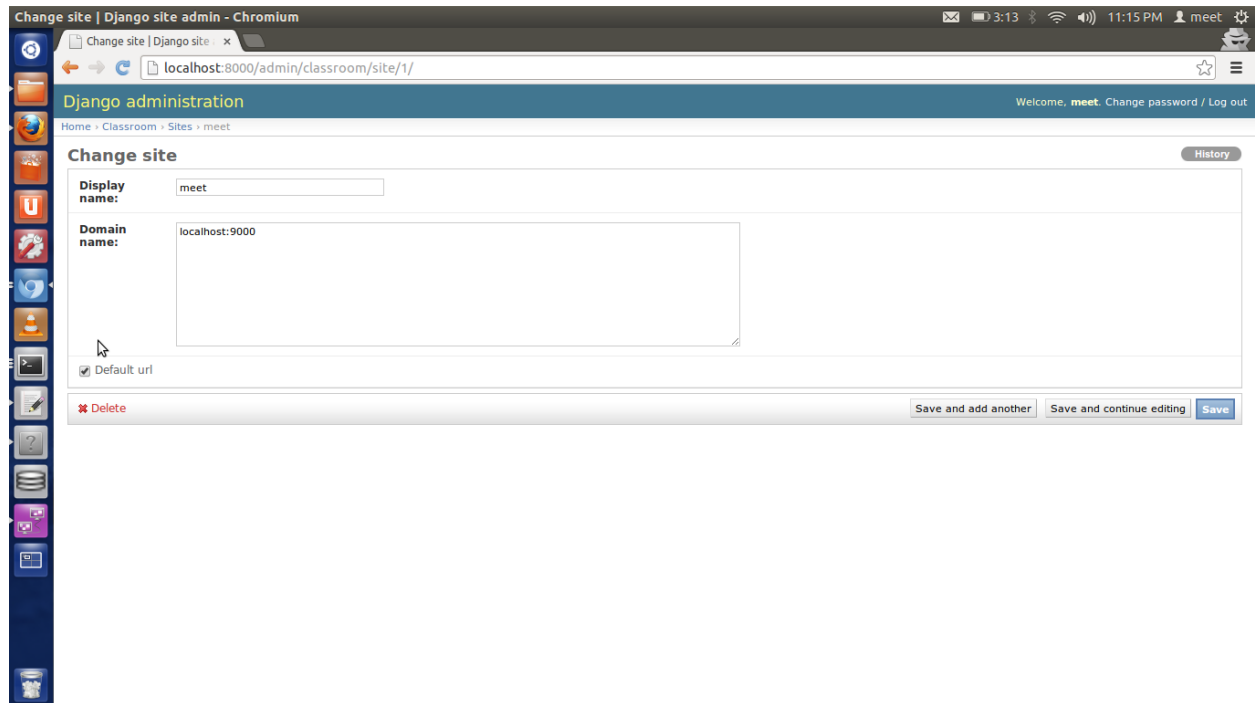


The image shows a web browser window displaying the Django administration login interface. At the top, there is a blue header bar with the text "Django administration" in yellow. Below the header, the login form is centered. It contains two input fields: "Username:" with the text "meet" entered, and "Password:" with four dots (masked) entered. Below these fields is a "Log in" button.

2. Select Classroom/sites



3. Add your service by entering groups' name i.e. your MOOC's name and IP address.



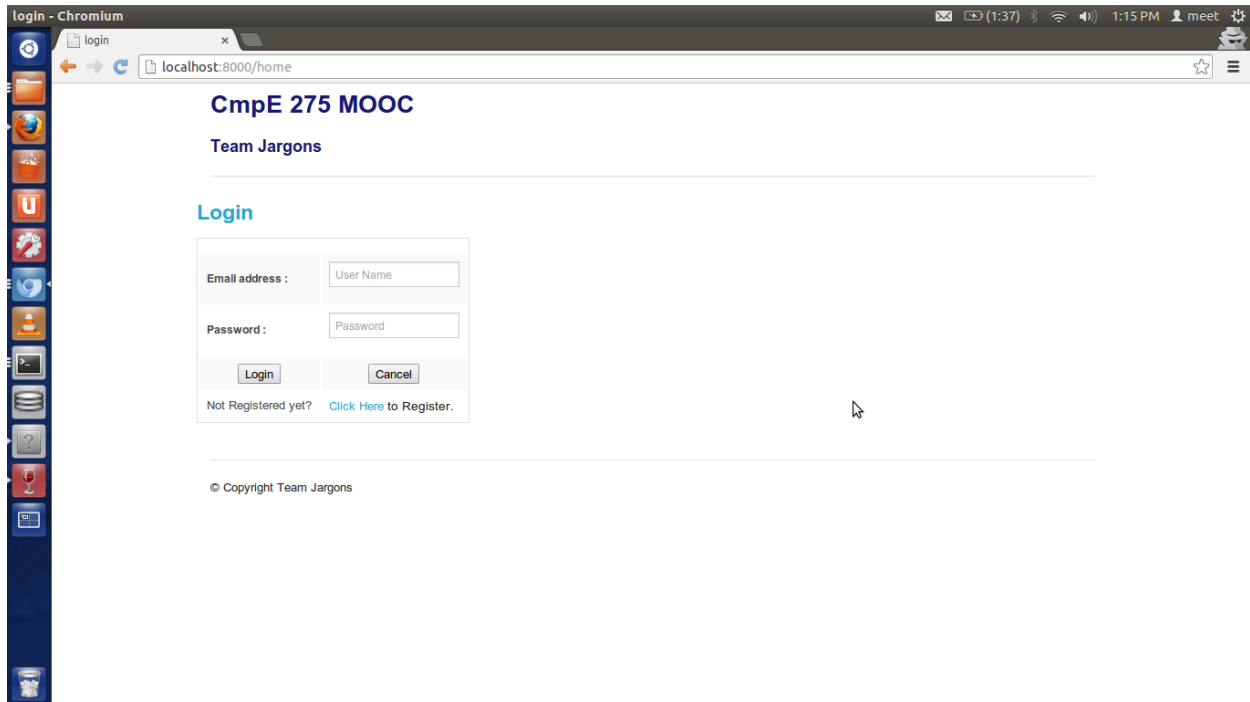
4. Click it as a default URL and save it.

5. Likewise, run the file which will execute the Bottle micro web framework which in turn will connect to the MongoDB database.

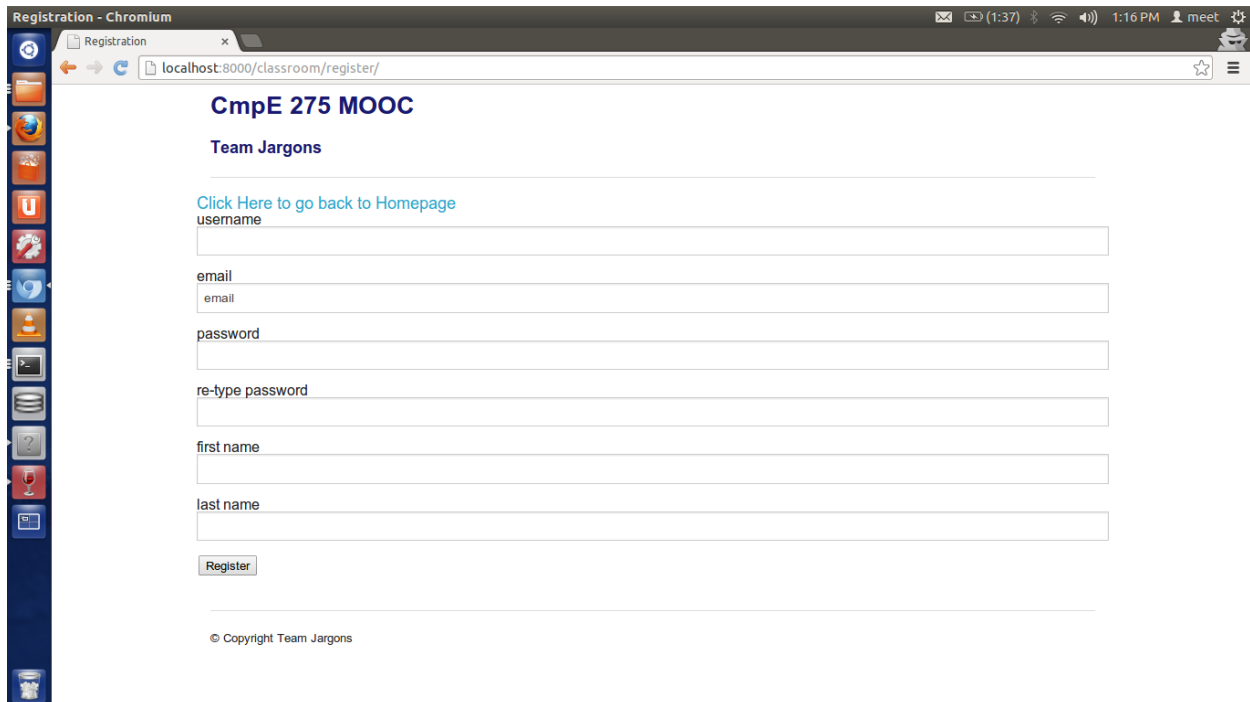
6. Now go to the path where the project resides and execute manage.py in terminal through

python manage.py runserver

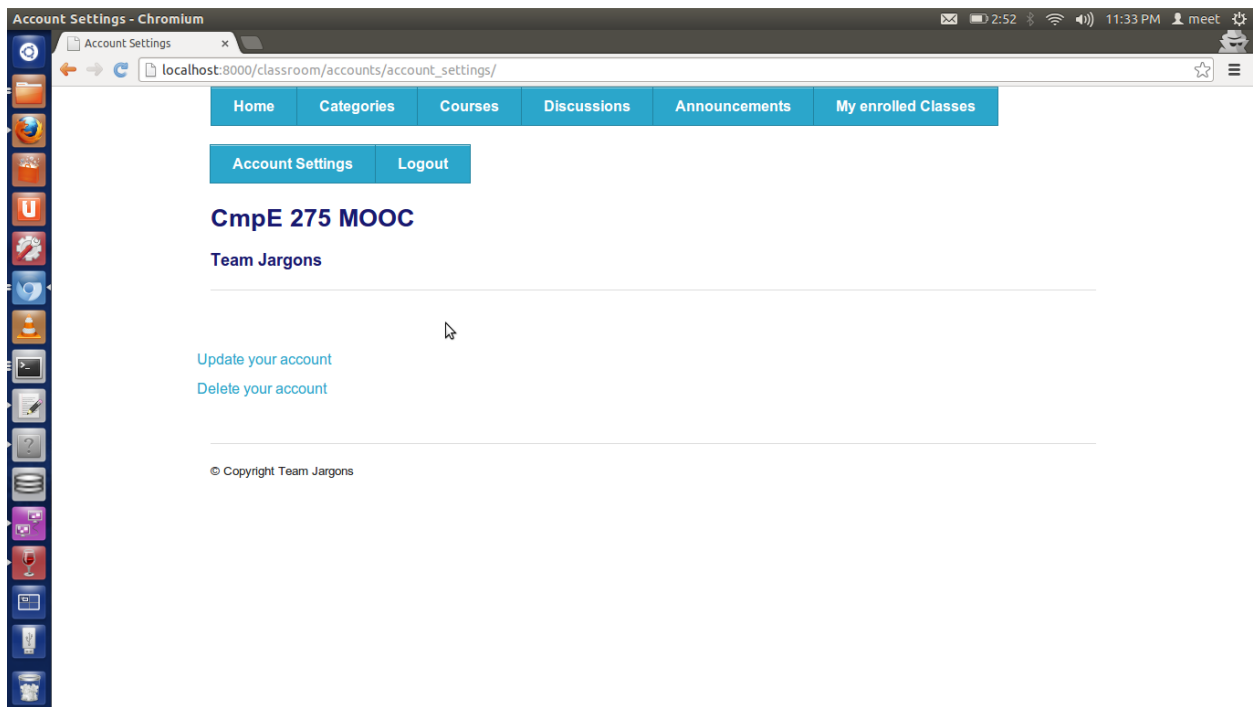
Screenshots



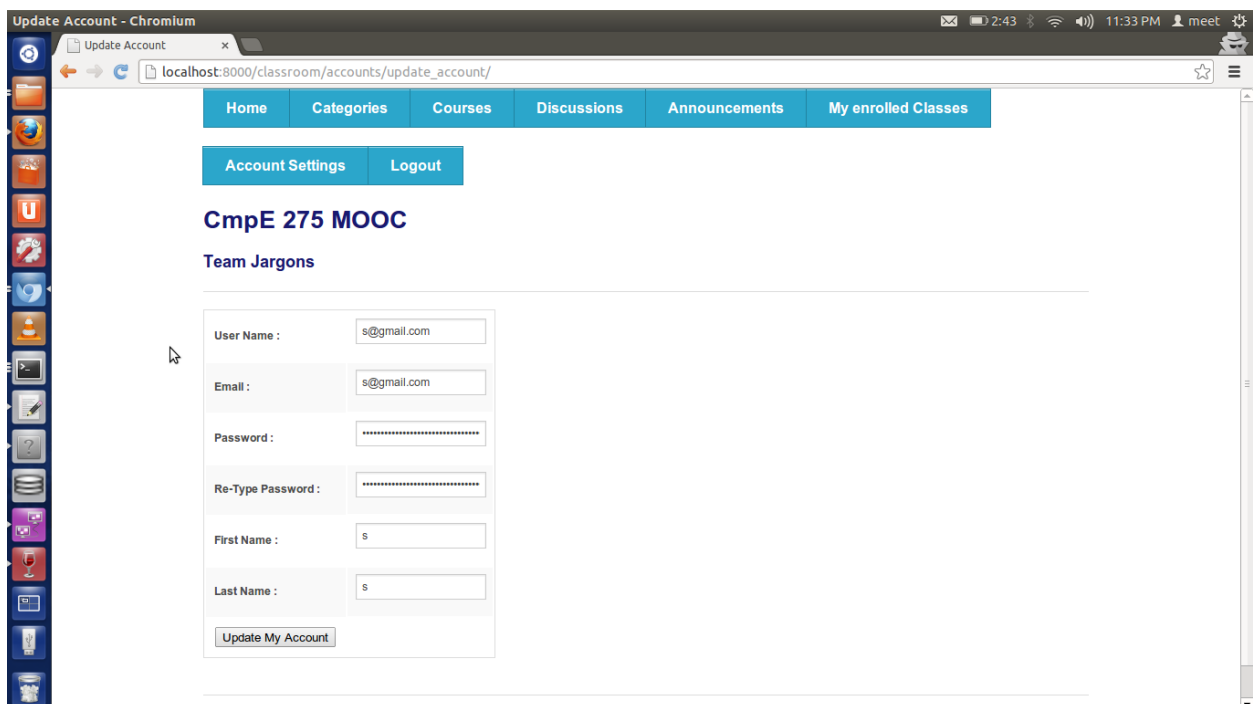
Home Page / Sign In



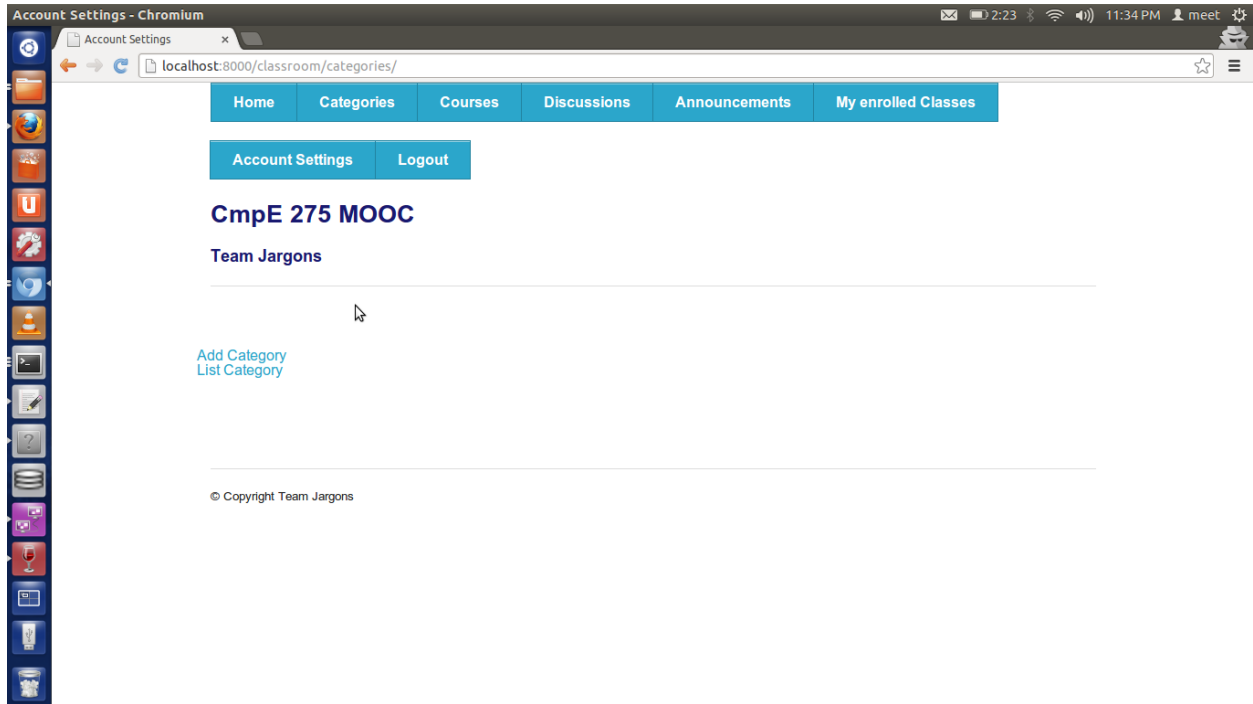
Register



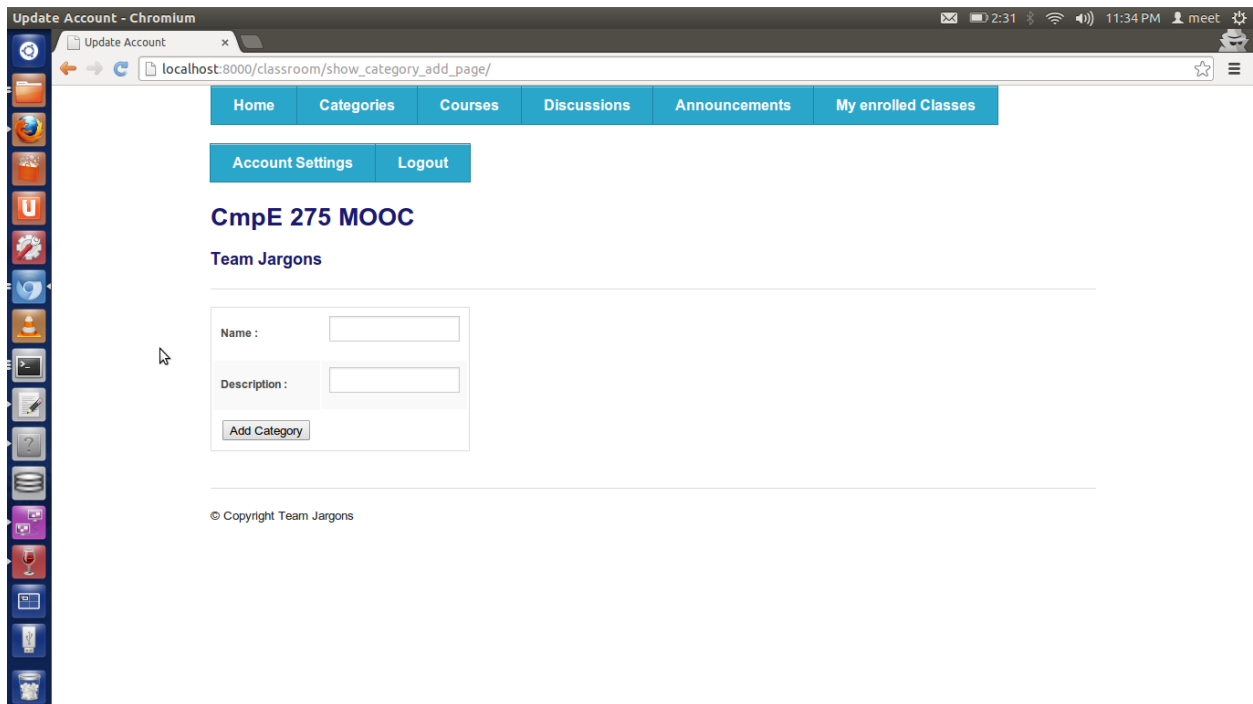
Click on Account Settings



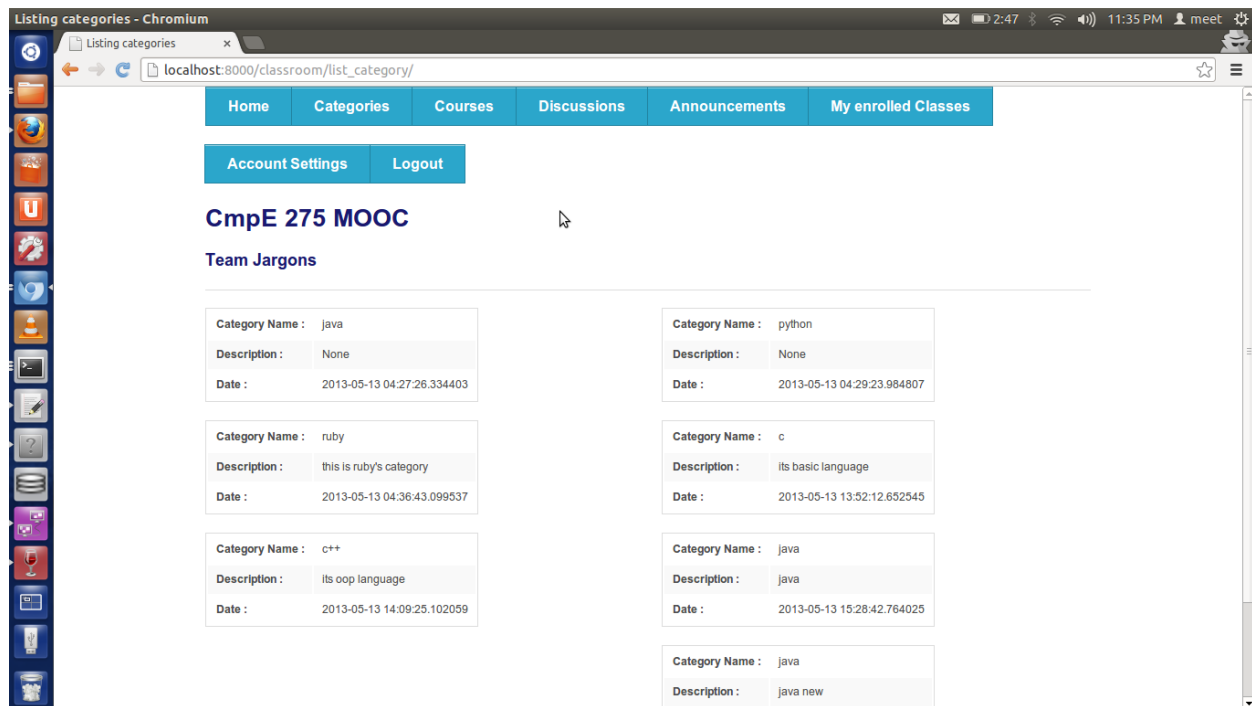
Update



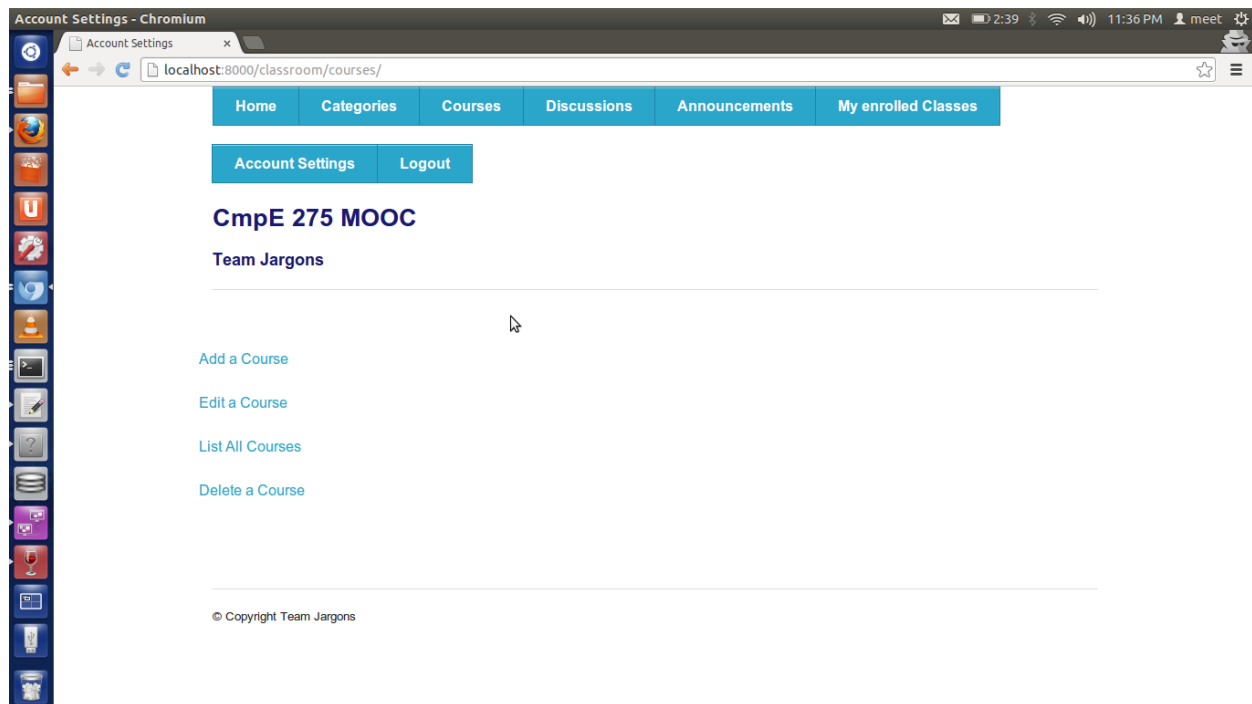
Category



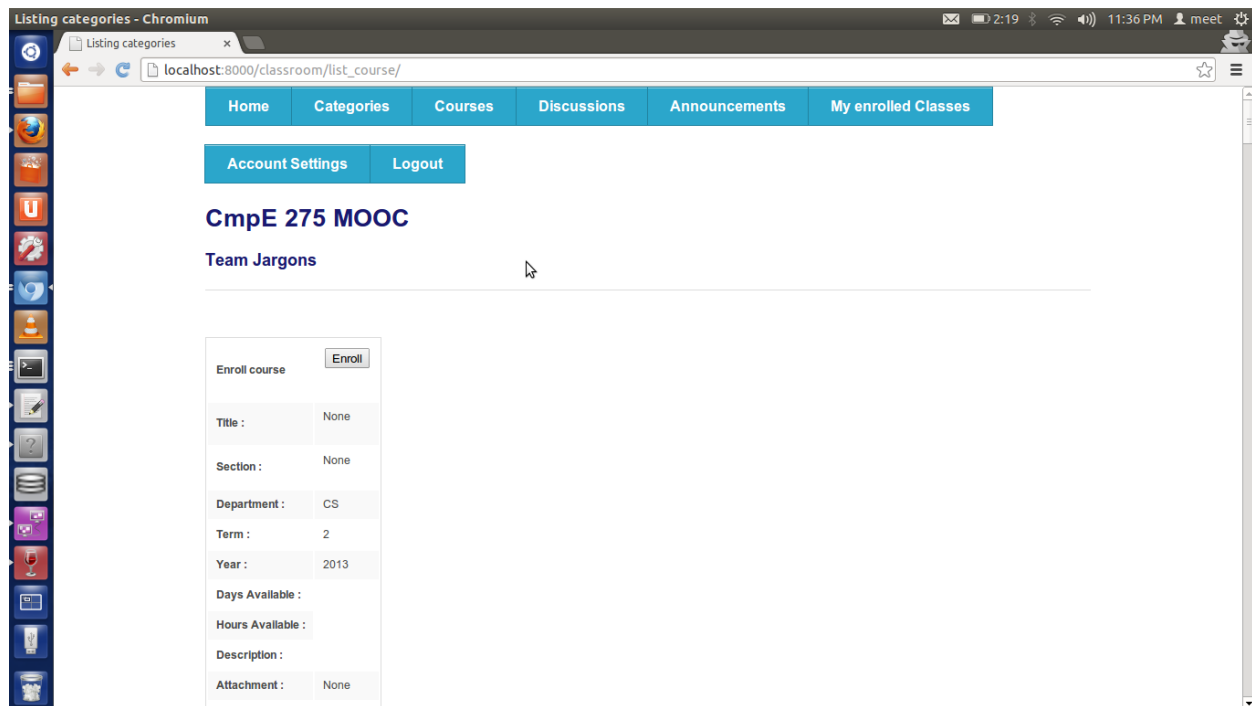
Category Add



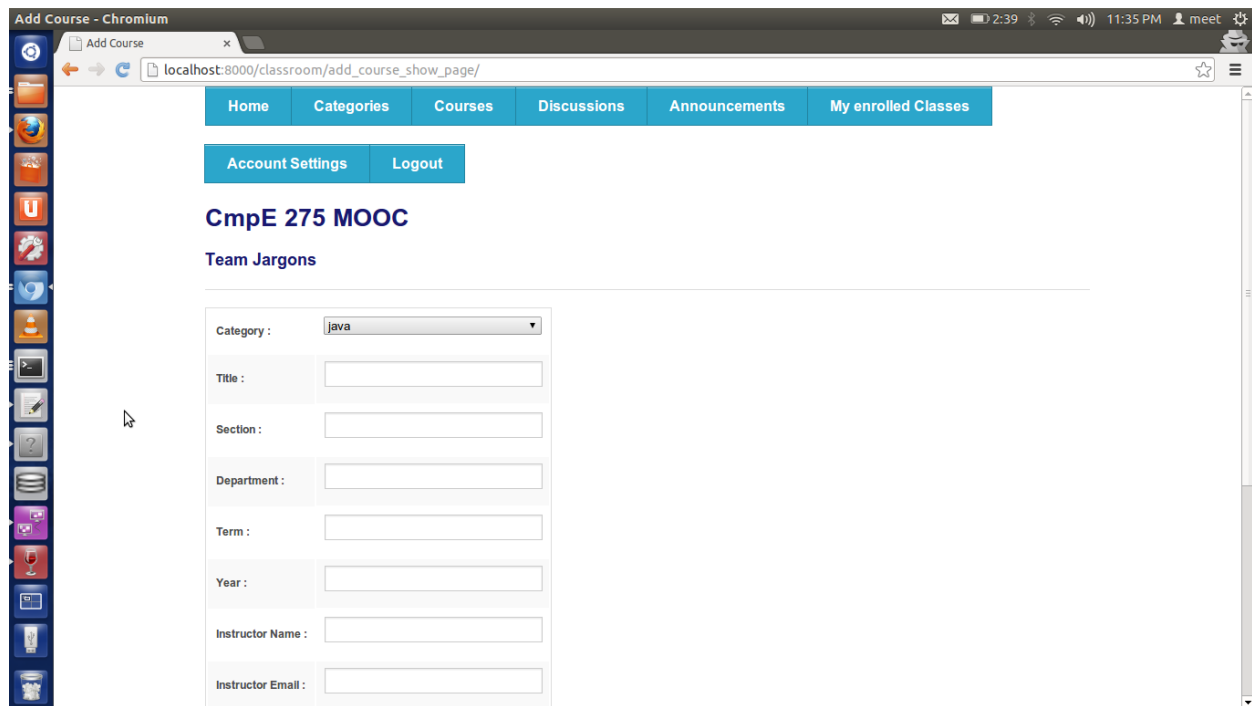
Category List



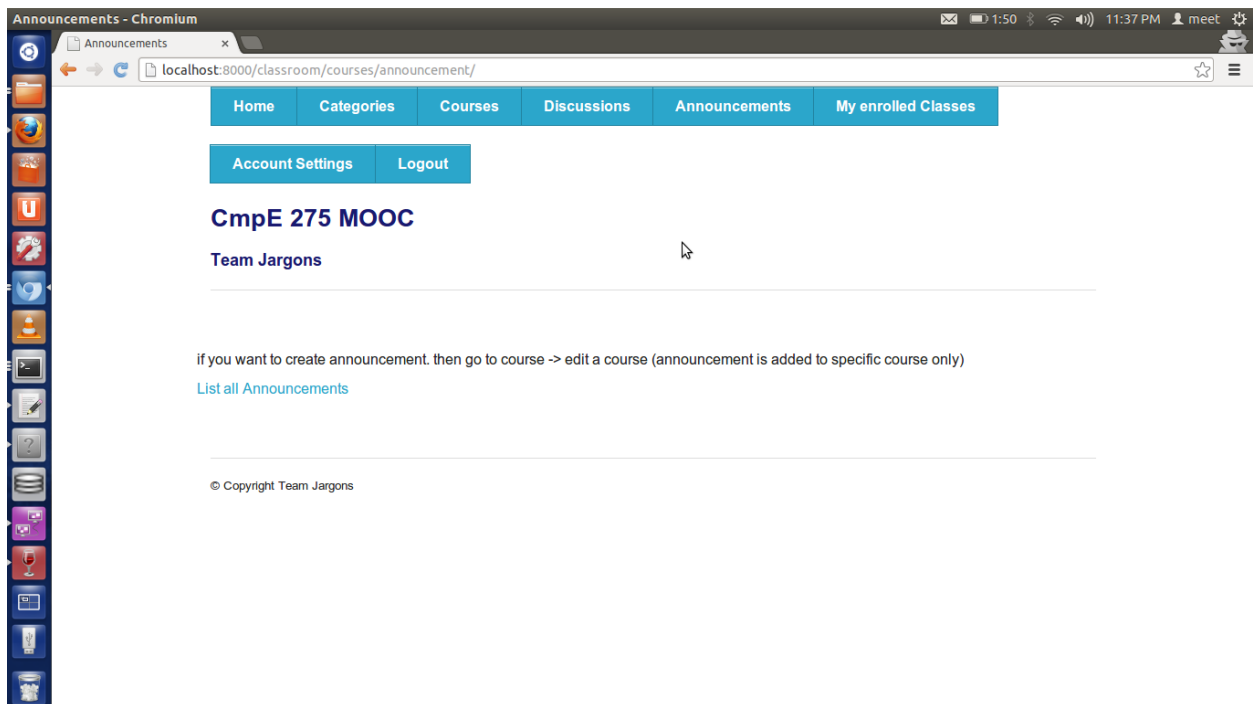
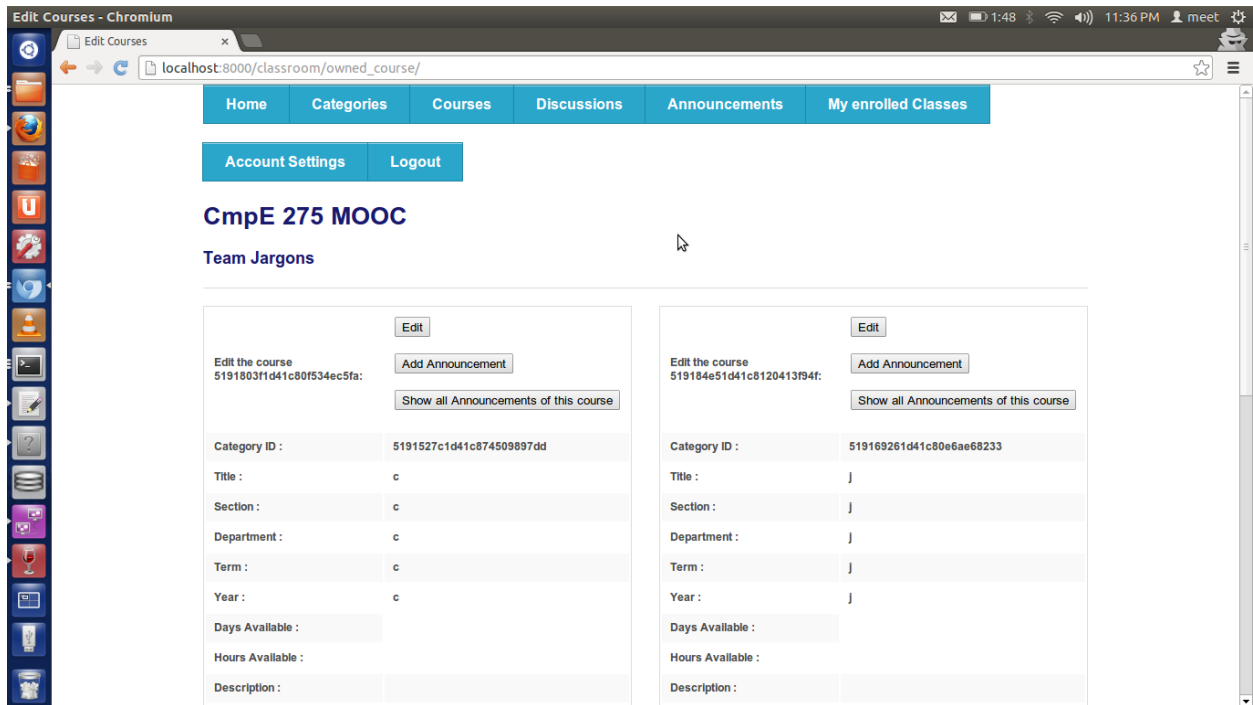
Course

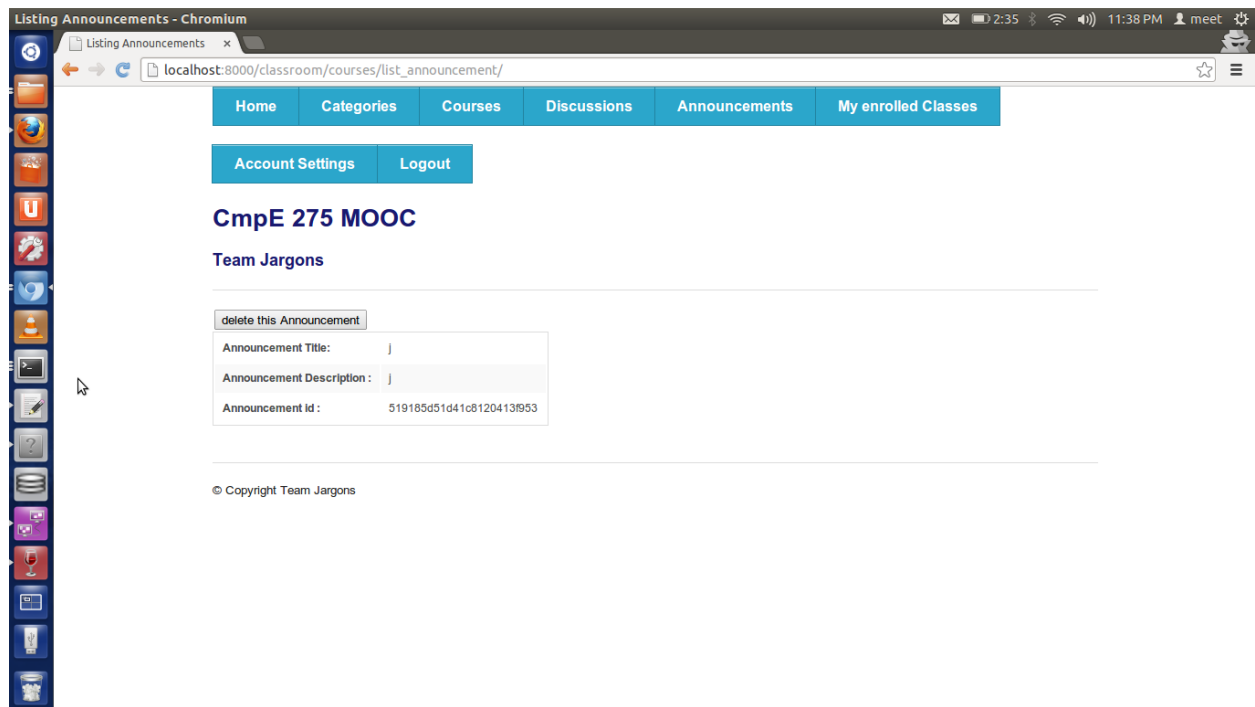


Course List

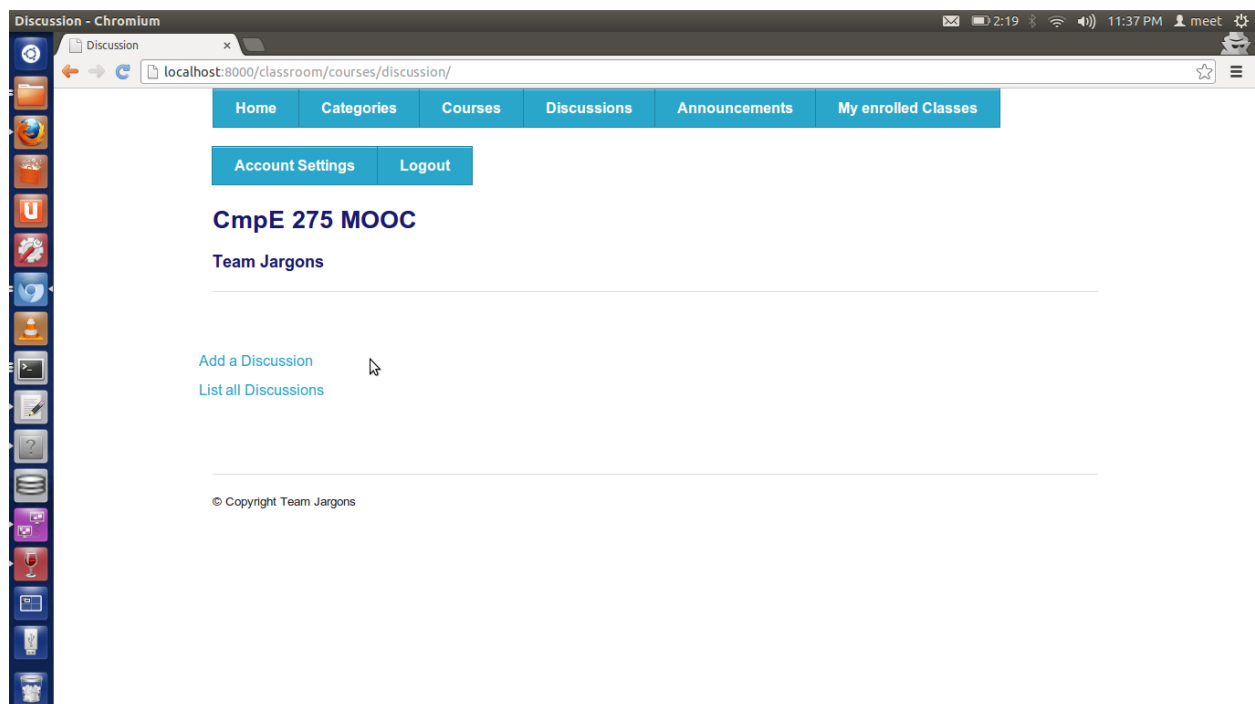


Course Add

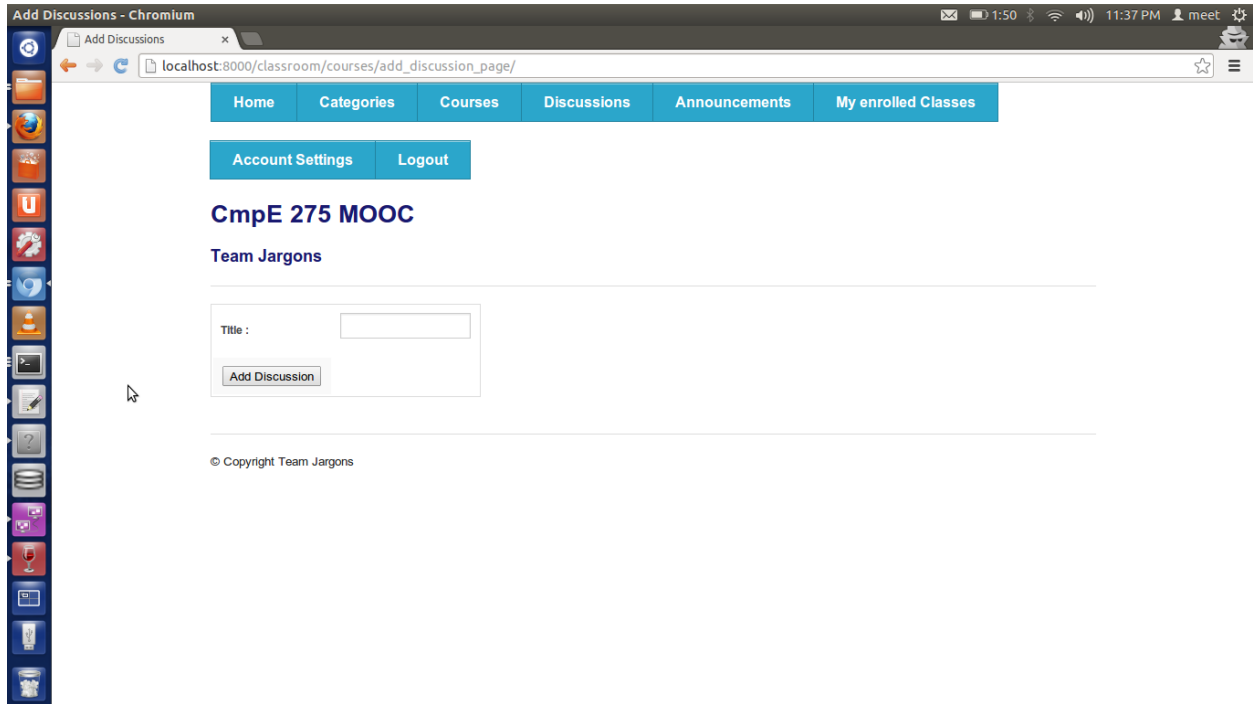




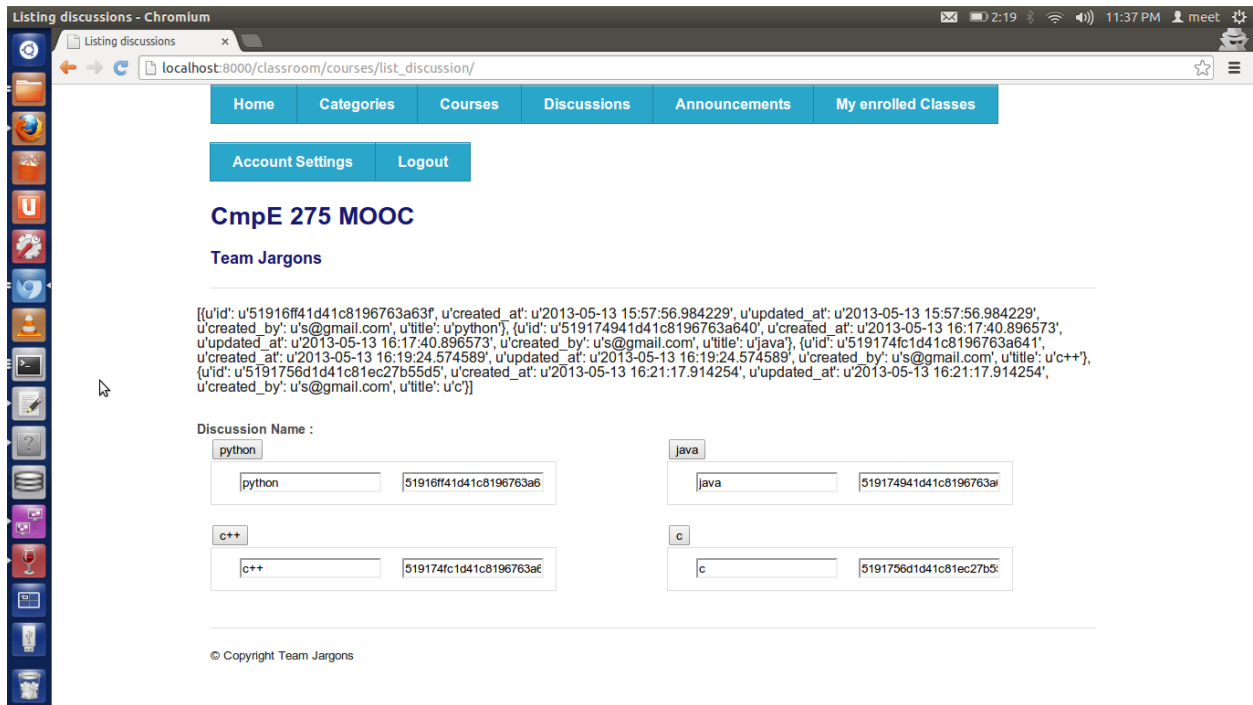
Announcement List



Discussion



Discussion Add



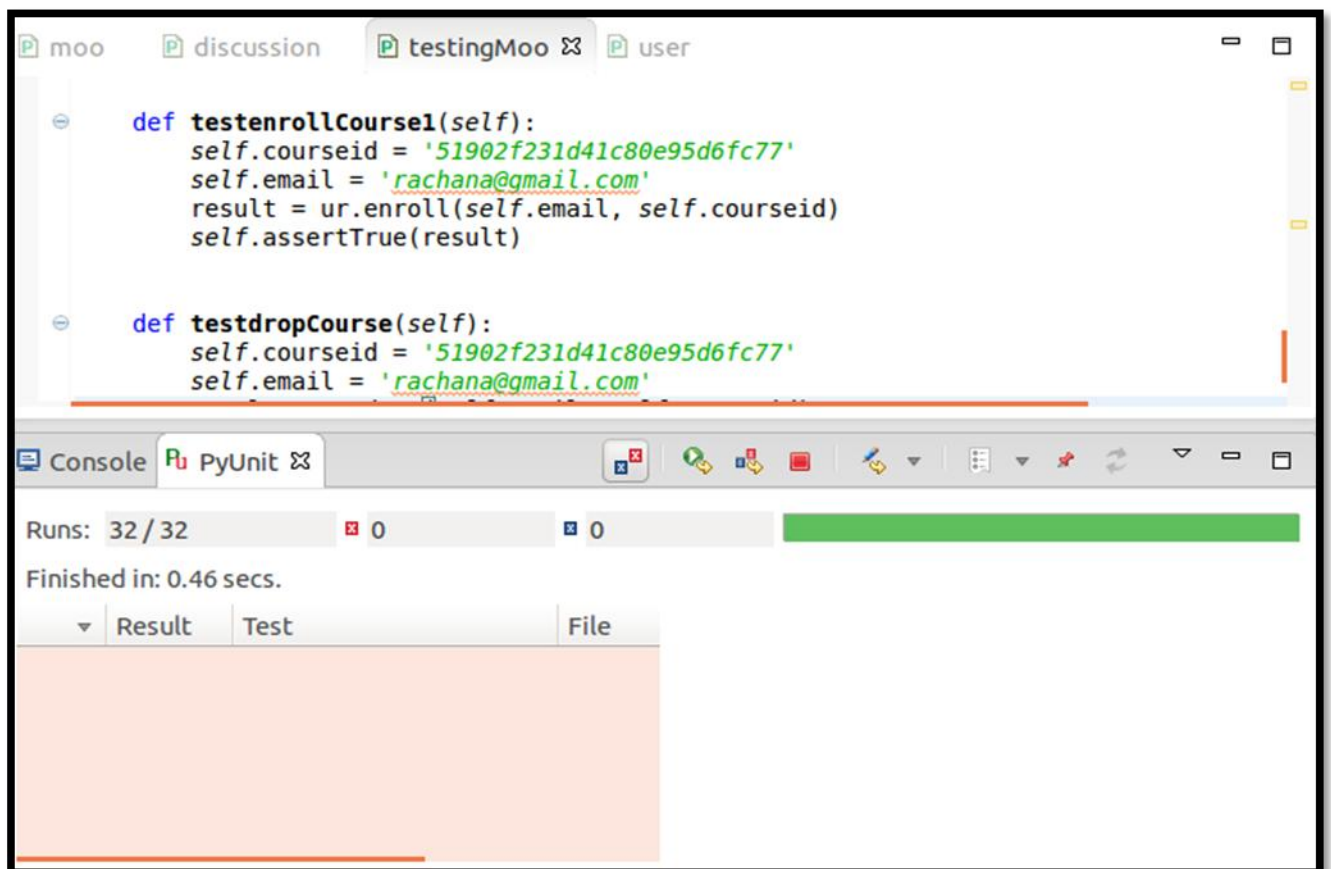
Discussion List (not perfectly seen as template is not loaded properly but data is correct)

MOOC Testing using PyUnit

Testing is one of the development phases of every project

t and hence we decided to create test cases for MOOC project. Since python was a new language for us and we were familiar with JUnit test cases, we used python unit test cases for testing.

PyUnit is the testing framework for python projects which is similar to JUnit of Java version. Before writing test cases we need to tell the python interpreter that this class contains test cases and should be executed as Python unit-test. To do this we need to create a subclass of “unittest.TestCase”. We wrote test cases for methods of MOOC interacting with the mongodb database. We add data explicitly in the test cases which will then be passed to the method that needs to be tested. Then we check the expected results of the methods with assert functions. If the expected result matches the actual result then the test case will pass else if expected result doesn't match the actual result then the test case will fail and raise an error.



The screenshot shows an IDE with a file explorer at the top containing 'moo', 'discussion', 'testingMoo', and 'user'. The 'testingMoo' file is open, displaying two test methods:

```
def testenrollCourse1(self):  
    self.courseid = '51902f231d41c80e95d6fc77'  
    self.email = 'rachana@gmail.com'  
    result = ur.enroll(self.email, self.courseid)  
    self.assertTrue(result)  
  
def testdropCourse(self):  
    self.courseid = '51902f231d41c80e95d6fc77'  
    self.email = 'rachana@gmail.com'
```

Below the code editor, the 'Console' and 'PyUnit' tabs are active. The console output shows:

Runs: 32 / 32 0 0

Finished in: 0.46 secs.

Result	Test	File

Things not used from Django:

- We had not used UserCreationform because we ran into errors and instead we maintain the form by our own.
- We had also not used csrf so we import exempt_csrf because we ran into errors because of it.
- We had not used models to fetch domain_url completely (we used it but not in all functions)

Project Status:

We almost completed all functionality but functionalities like drop course, quiz, add message to discussion is remaining.

Contributions:

Meet: Django admin configuration, sites decoupling (enter url through admin interface), django's interaction with *RESTful* API and implementing its different views.

Neel: GUI, Templates implementation, static files including CSS & JS inside it

Rachana: Implementation of test cases using pyunit and checking all RESTful APIs working properly

Snehal: Implementation of *all RESTful APIs*(Bottle, MongoDB) , Creation of new API document to define the requests and responses.

References:

1. http://en.wikipedia.org/wiki/Massive_open_online_course
2. <http://bottlepy.org/docs/dev/>
3. https://en.wikipedia.org/wiki/Representational_state_transfer
4. http://en.wikipedia.org/wiki/Web_API
5. <http://docs.mongodb.org/manual/reference/sql-comparison/>