**Project Report**

**On**

**Student Registration System**

****

CMPE 273 Spring 2013

Section 2

Manjunath Shivanna:008623368

Harshal Sakpal:008646183

Shashank Bharadwaj: 008673626

Neha Thiyagarajan: 008123232

Veenu Agarwal: 008718541

# 1. INTRODUCTION

The goal of this project is to create a Student registration system to offer online course registration system to students. The system offers Online course registration system for students. Student can log in and search for a course, add or drop the course. Professor can view the students who have registered for the courses and relevant details.

The system has been implemented using Web Services. Optimization techniques like connection pooling and object caching have been used to increase performance, scalability and reliability of the system. Test cases have been developed using J Unit to ensure that the system works in all scenarios. J Meter has been used to measure and compare the performance of different optimization techniques used.

**Contribution**

**The entire team decided database design and the architecture.**

**Manjunath Shivanna :** Client and server implementations of add Course, drop Course, Server and client side validations for sign up page. Implemented connection pooling, in memory caching, batch processing. Test stubs written to generate random load for testing different scenarios like caching, connection pooling etc. Scripts written to load data. End to end integration and testing.

**Harshal Sakpal :** Database designSearch course by all categories, Sign Up , Sign In Main page for professor and Student, End to end integration and testing.

**Shashank Bharadwaj:** Billing, Integration and end to end testing, search and display all members based on category, also add course, report,J meter Testing End to end integration and testing. Test graphs for Connection pooling, Object caching and batch processing.

**Neha Thiyagarajan:** Login, add course**,** Update Course and Update Student, Update password, Junit Test cases, J Meter testing, Client Side Validations, Code Integration, End to end testing, also return course

**Veenu Agarwal:** View all courses rented by user, View all users who have rented a course , server side validations, Server side validations , End to end integration and testing.

# System outline

The system operates in two modes: student and **professor,** controlled through login**.** Each mode supports a different set of functionalities.

## Functional Requirements

#### User Perspective

1. Create and delete Student account.
2. Search courses based on category
3. Edit account details
4. View his currently registered courses
5. View his transaction history
6. View and pay bill

***professor*** ***Perspective***

1. Create, update, delete accounts
2. Search Students, courses based on category(Includes listing all courses/members known to the system)
3. Add, update, delete courses
4. View transaction history of all users and courses
5. View list of courses registered by user
6. View all students registered in a particular course

***Other*** ***features***

1. Bill generation

**Non Functional Requirements**

* Reliability: The system should be reliable enough. Even an error occurs the user should be informed
* Performance: The system should be available all most of the time.
* Accuracy: The system should be accurate that is perform all the calculations correctly.
* Scalability: The system’s performance should not degrade with increase in load.

## Object Management

The following object entities are maintained using POJO. All the object entities correspond to database tables where their states are stored for persistence.

|  |  |  |
| --- | --- | --- |
| Object | DB Table | Object Content |
| Student | Student | firstName, lastName,sjsu id  emailId, password, address, city, state ,zipCode, gender, password, role  balanceAmount, state name |
| Course | Course | CourseNo, courseName, Section, Credits, Department, Day, Time , Location |
| Professor | Professor | firstName, lastName,sjsu id  emailId, password, address, city, state ,zipCode, gender, password, role  state name |
| Transaction | Transaction | IdTransaction, idCourse, issueDate |

## Services

The web services form the middle tier of the system. This layer holds all the business logic of WSDL of these services is published by the server. The client can call any of these services using the endpoint made available through WSDL.

## Update policy

Accessing and updating a database state is an expensive operation. Hence database updates must be done only if the value has changed since it was previously fetched.

## To handle this, we have implemented caching (using synchronized Hash Map). This cache will always be in sync with the database tables. Whenever there is an update request, the cache is checked and the new values given by user are matched with the ones in cache. If there is no difference between the two, then there is no need for update, hence saving a trip to the database. If there is a difference between the two, the database is updated.

## Heavy weight resource handling

Every system should be highly scalable. The system's performance should not degrade as the load on the system increases. The performance will start degrading if heavy resources are not handled properly. We should try creating heavy resources only as much is required and reuse most of them.

We have handled heavy resources using the following strategies:

* Connection Pooling
* In memory Caching
* Batch Processing
* Use of prepared statements
* Rollback and auto commit of transactions

The following graphs show the how performance has been enhanced using the above. The scenarios have been tested as those which will be the most frequently used by the users:

**Adding a course**

Figure 1 Add course (tested for 1000, 10000,100000 courses)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number of insert Queries | Simple Query(ms) | Prepared Statement(ms) | Connection Pool (ms) | Batch (ms) |
| 1000 | 14.115 | 14.303 | 3.483 | 0.34 |
| 10000 | 113.220 | 115.2 | 34.013 | 1.79 |
| 100000 | 1062.22 | 1073.240 | 337.005 | 16.820 |

The above graph shows the difference in performance in adding courses based on following four criteria:

Simple Query, Prepared Statement, Connection Pool, Batch Inserts. The above scenarios have been tested for adding 1000, 10000, 100000 courses.

As we see in the above graphs **simple statements** show the worst performance while adding courses in batch show the best performance.

Simple statements are not precompiled.

**PREPARED STATEMENTS**

These are precompiled. System first parses the statements for syntax errors and then finds an optimal way for executing the statements particularly that involving multiple tables. Also it becomes faster if you execute the same statement again and again with different parameter values as the statement does not need to get compiled every time for each execution.

**CONNECTION POOLING**

We have implemented connection pooling using a queue.

Queue size is maintained between MAXPOOLSIZE and MINPOOLSIZE.

When the system starts a pool of MINPOOLSIZE of 30 connections to the database are created. As an when client request comes connections are given out from the pool. If there are new connection requests and all the connections in the pool are busy then more connections to the database are created. But there is a limit to the maximum connections which we have set to 40 in the pool. If that limit is reached then the client has to wait for some connection to become empty.

For ending connection then the current connection is returned to the pool.

We see that the time taken to insert course decreases by a major amount of we use connection pool. This is because every time when a course is inserted the client does not have to wait for the connection to be created. The connections are given out from the pool. So no time is wasted to make connections.

But if we do not use connection pool then every time a course is added a new connection to the database is created. Creation of a new connection is a very heavy operation so connection pooling saves CPU cycles and latency.

**CACHING**

Figure 6 Search Course Caching/Connection Pooling

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Scenarios | Cache size 1000/100000 | Cache size 75/100000 | Cache size 1000/10000 | Cache size 75/10000 |
| NoCache/no Connection pool | 76091 | 76924 | 12062 | 12111 |
| No Cache/Connection pool | 8329 | 7853 | 1024 | 1034 |
| Cache/NoConnection Pooling | 7560 | 22677 | 1169 | 3244 |
| Cache/Connection pooling | 2866 | 3038 | 308 | 362 |

The above graph shows the performance of searching a course under the following scenarios:

* No caching and co connection pooling
* No caching and with connection pooling
* Caching and no connection pooling
* Caching and connection pooling

The above scenarios are tested for cache sizes of 1000 and 75 and for 100000 and 10000 courses in database.

**Caching implementation**

We have implemented an in memory Least Recently Used (LRU) cache which is a subtype of LinkedHashMap. We used LinkedHashMap as it is easy to implement LRU caching with it. In LRU caching we should remove the least used entry in the cache when the cache is full.

This can be easily done by overriding the removeOldestEntry function of the LinkedHashMap.

The rest of the functions like put, get are similar to hash maps. Using hash maps is good for caching as searching is an O(1) operation in hash maps.

To get the actual performance of the database all these scenarios have to be executed under the same condition and with the same load.

To simulate real world query requests we create a random list of numbers and a list of course names. The random list of numbers is index into this list of course names.

So each of the search queries iterates through these random numbers and calls the search with course names present at that index.

This way we know that we are calling each scenario with a similar set of queries.

**Results**

No caching and no connection pooling show the worst performance. For every query a connection to the database is made.

With no caching the database is hit for every query.

With connection pooling and no caching , the results are better than the above scenario. This is because for every search request a new connection is not created. An existing connection form pool is taken to serve the request.

With no connection pooling and caching the performance is better than previous scenario as the database is not hit at all for the course that is found in cache. Only if the course is not found in the cache a new connection to the database is created to fetch course details.

Searching with caching and connection pooling gives the best results.

This is because for every search request a new connection is not created. Also if a course is found in the cache the database is not hit at all. This significantly increases performance.

Also if we compare caching performance with connection pooling and without it we see that the time taken decreases by almost one third times even though the number of cache miss and hits are the same in both the cases.. This is because a cache miss causes a database hit and that really increases the time taken without connection pooling.

We also compare the performance with a cache size of 75. With such a less cache size the number of cache misses increases. Therefore we see that the time taken to search courses with cache increases from 2866 msec to 3038 msec.

This increase in time can be seen in case of no connection pooling very significantly. We see that the time increases from 7560 msec for cache size of 1000 to 22677 msec for cache size of 75. This large increase time is because the number of cache misses increases from 100 (1000 cache size) to 286(75 cache size).

Thus we conclude that the best performance can be improved through connection pooling and caching subject to high cache hit rate. However if there are a lot of cache misses the performance might degrade.

**BATCH INSERTS**

Batch inserts show the best performance. By using batch we can add group of statements and execute the entire batch once. Since the database is hit only once for entering the entire batch of objects we get very high performance.

Batch processing improves performance in two areas:

1. If we give set of statements to the databases then it can optimize it to execute it most efficiently.

2. Secondly the network costs reduce as the number of round trips to the database reduces significantly.

So to conclude we should try and use batch inserts. But if we do not have a batch of courses to insert then connection pooling is the best option.

Also while testing if we add a lot of courses in a row without connection pooling we get a socket exception even when we close the connection. This is because the server has gone out of sockets. Sockets are not released with the same speed as we end connection and this throws exception. To avoid these kind of race conditions we should use connection pooling.

**Testing**

**Test Driven Development**

We have used test driven development. Following was the sequence of development:

* Test cases
* Server side implementations
* Client side implementations

Tests cases were written for each of the scenarios. The server side was implemented for the functionality. Then we checked each of the scenarios with different inputs.

Then the client was implemented after the server side functionalities were completely tested.

**Functionality Testing**

Junit

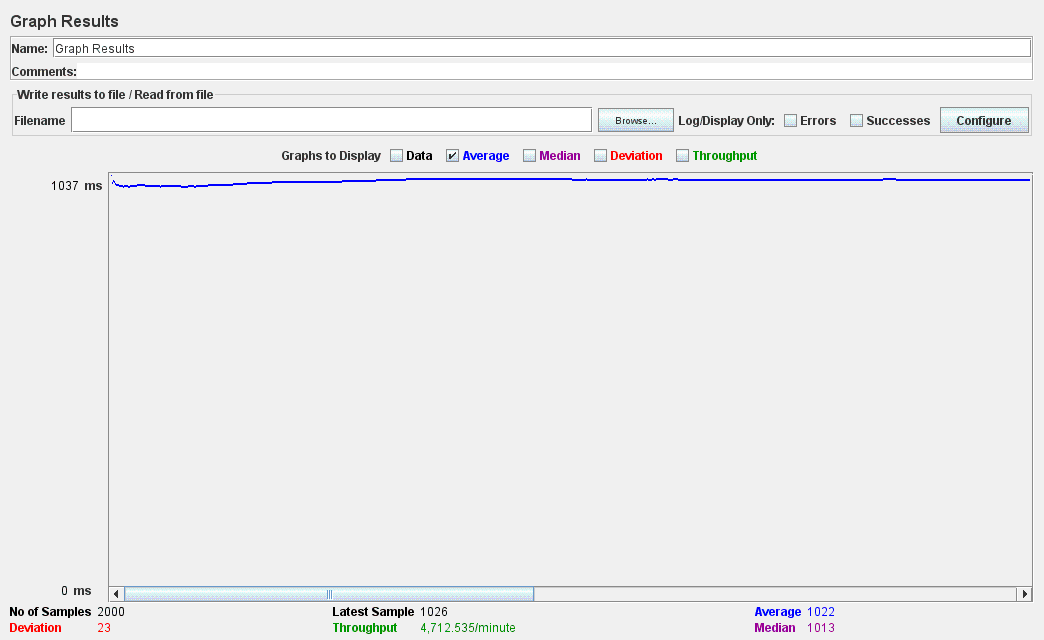
**Performance Test**

Sample test classes

Jmeter

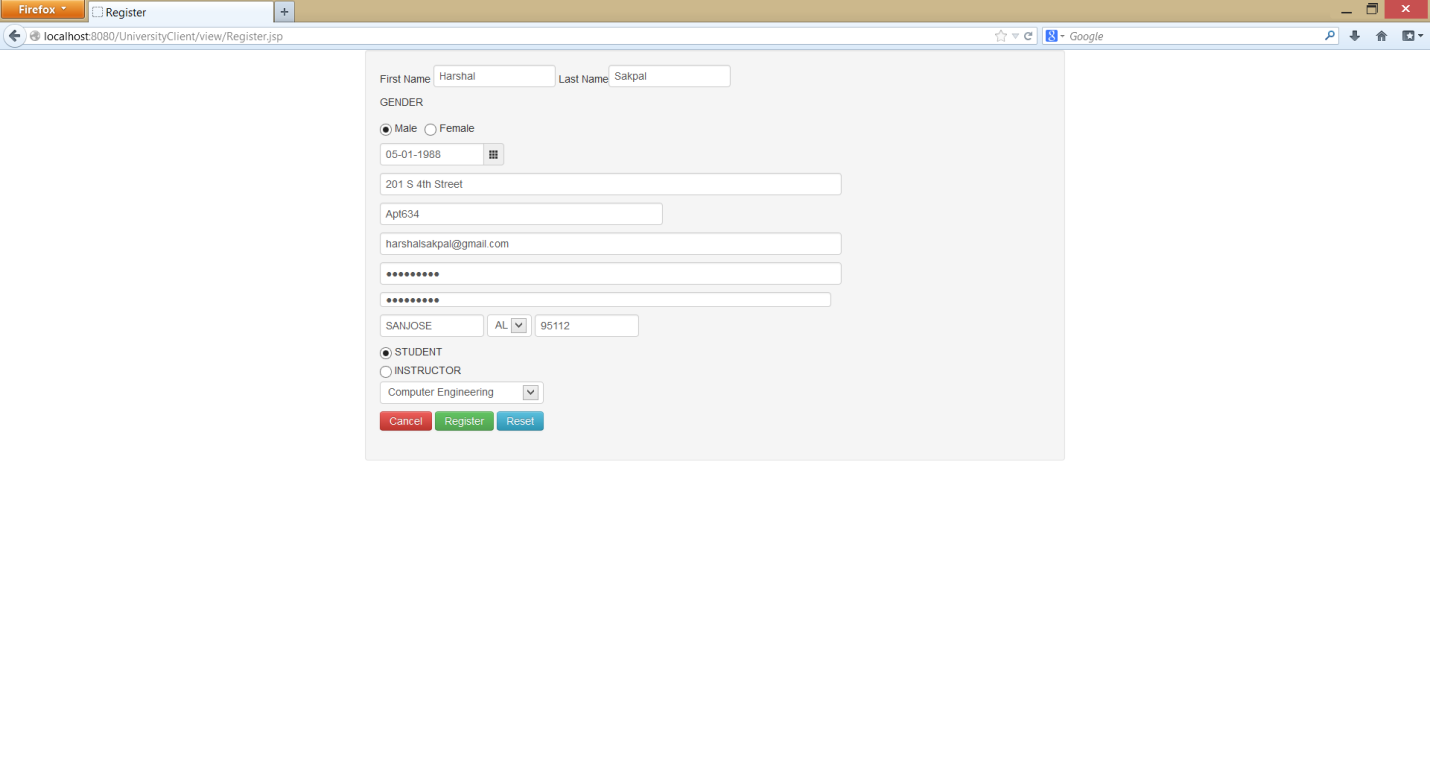
**Jmeter Performance results**

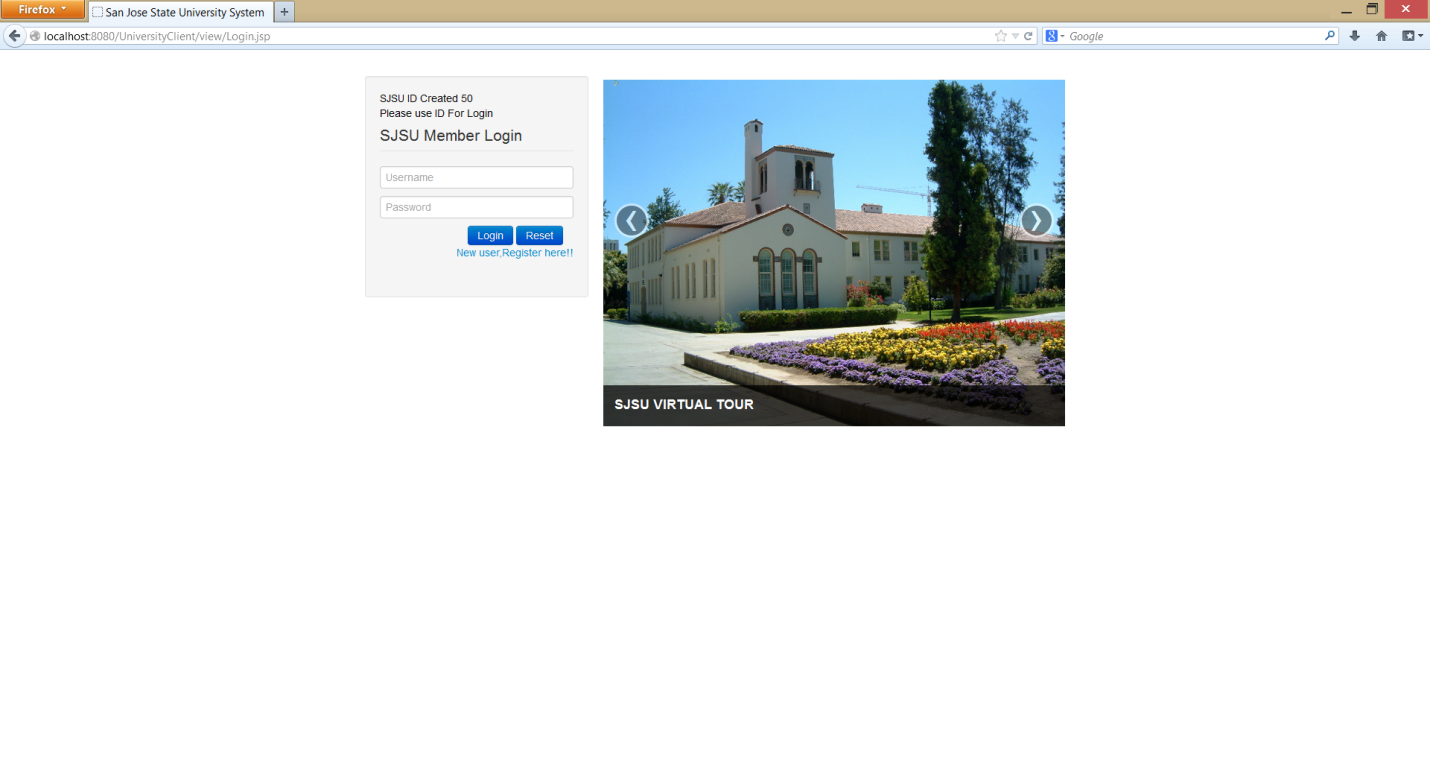
J Meter performance for insertCourse() function with connection pool

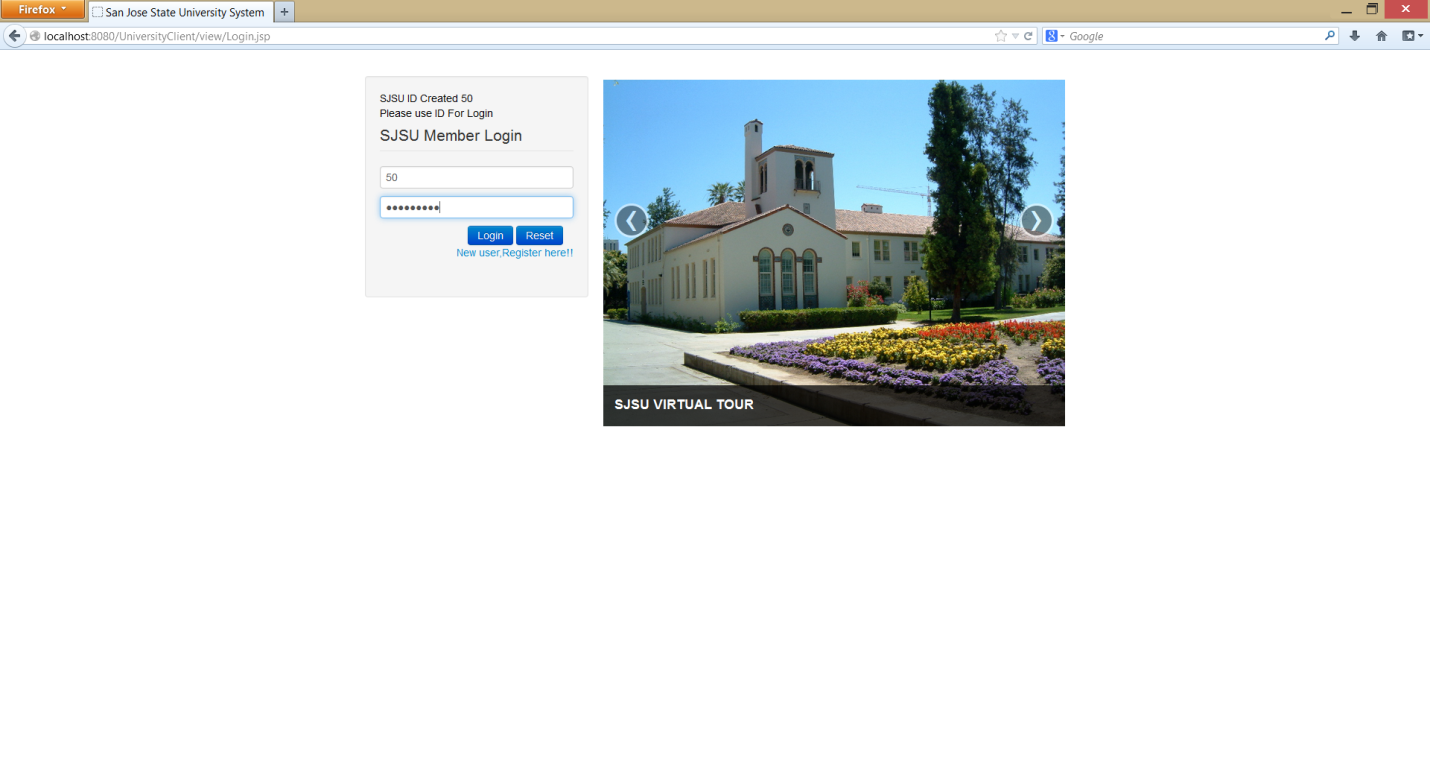
****

**Project Screenshots:**

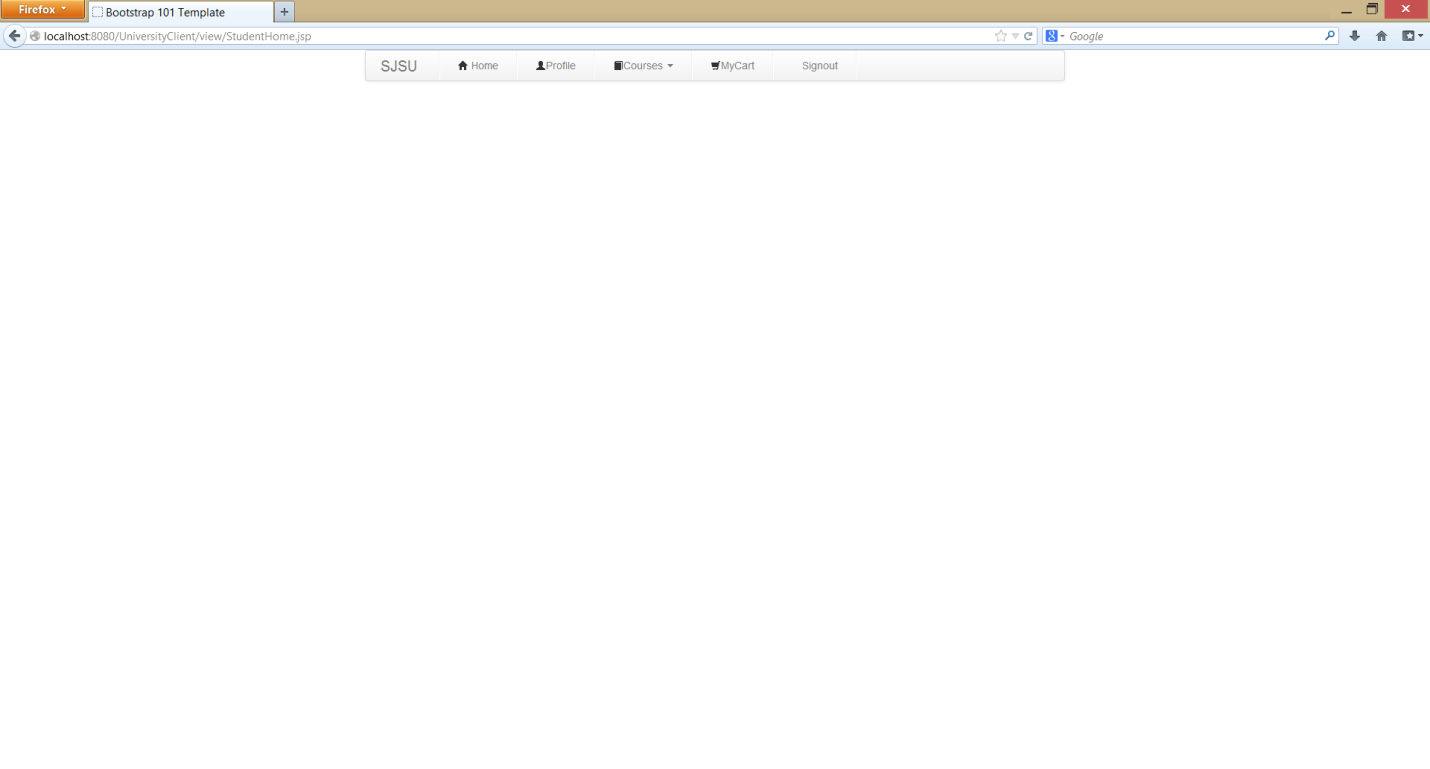
**Member Registration:**

****

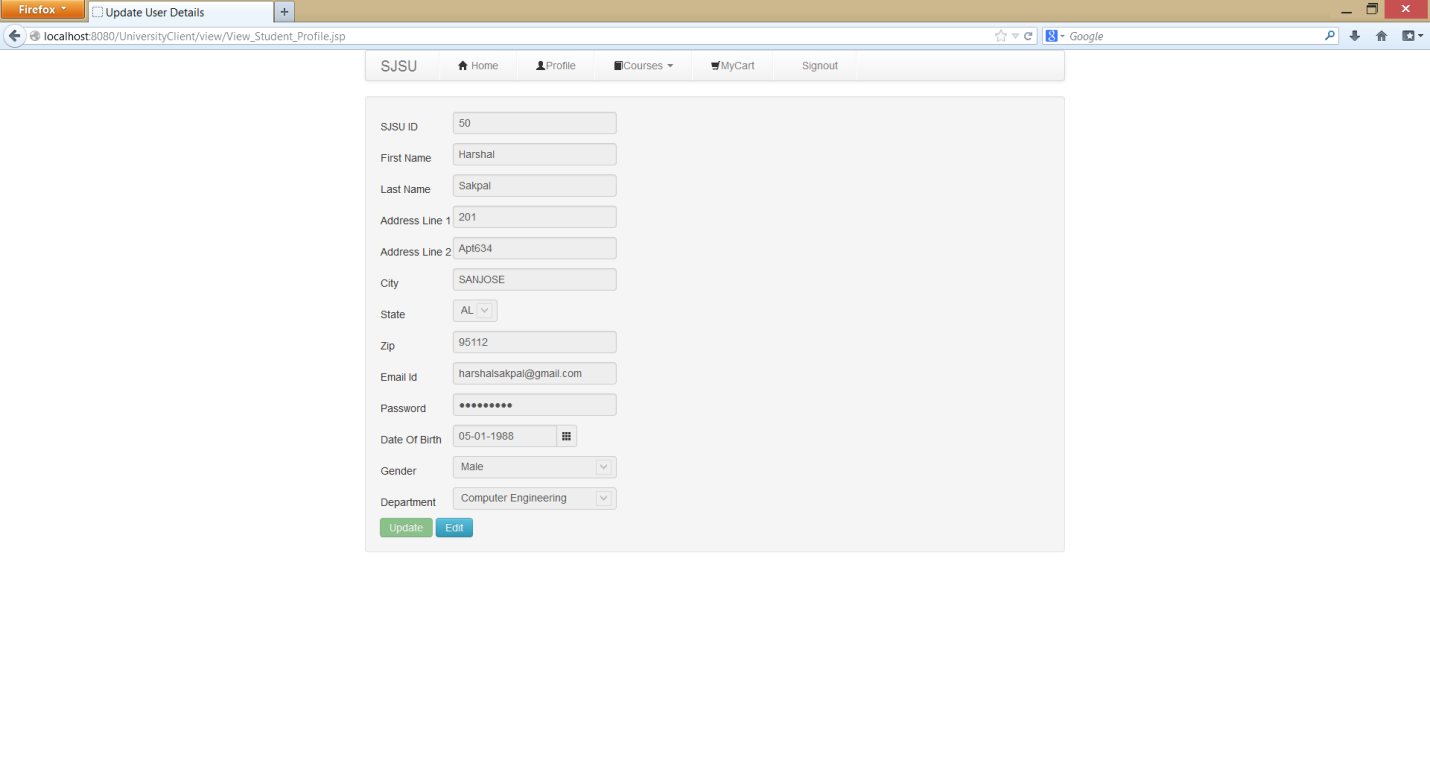
****

**Member SignIn:  
**

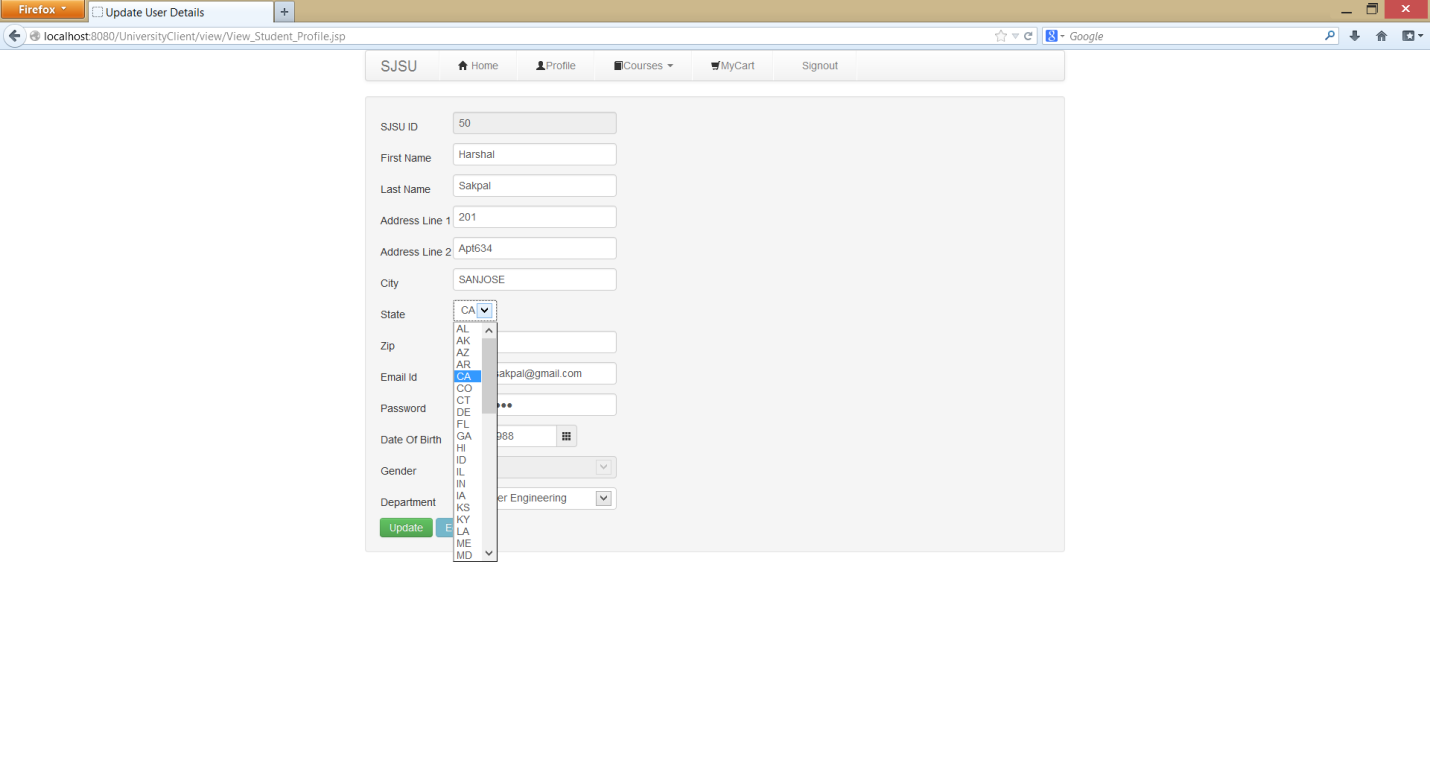
**Member Home:**

****

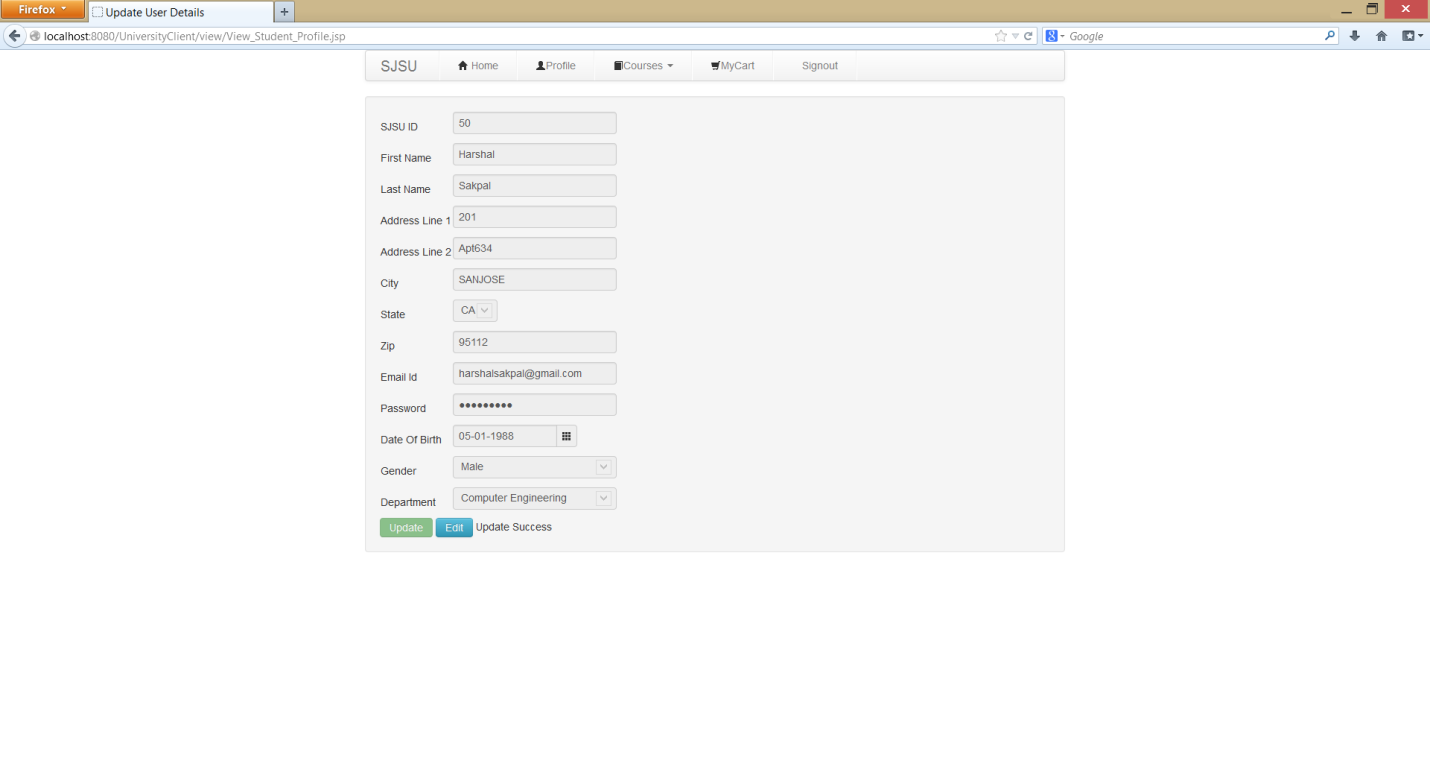
**Member Profile View:**

****

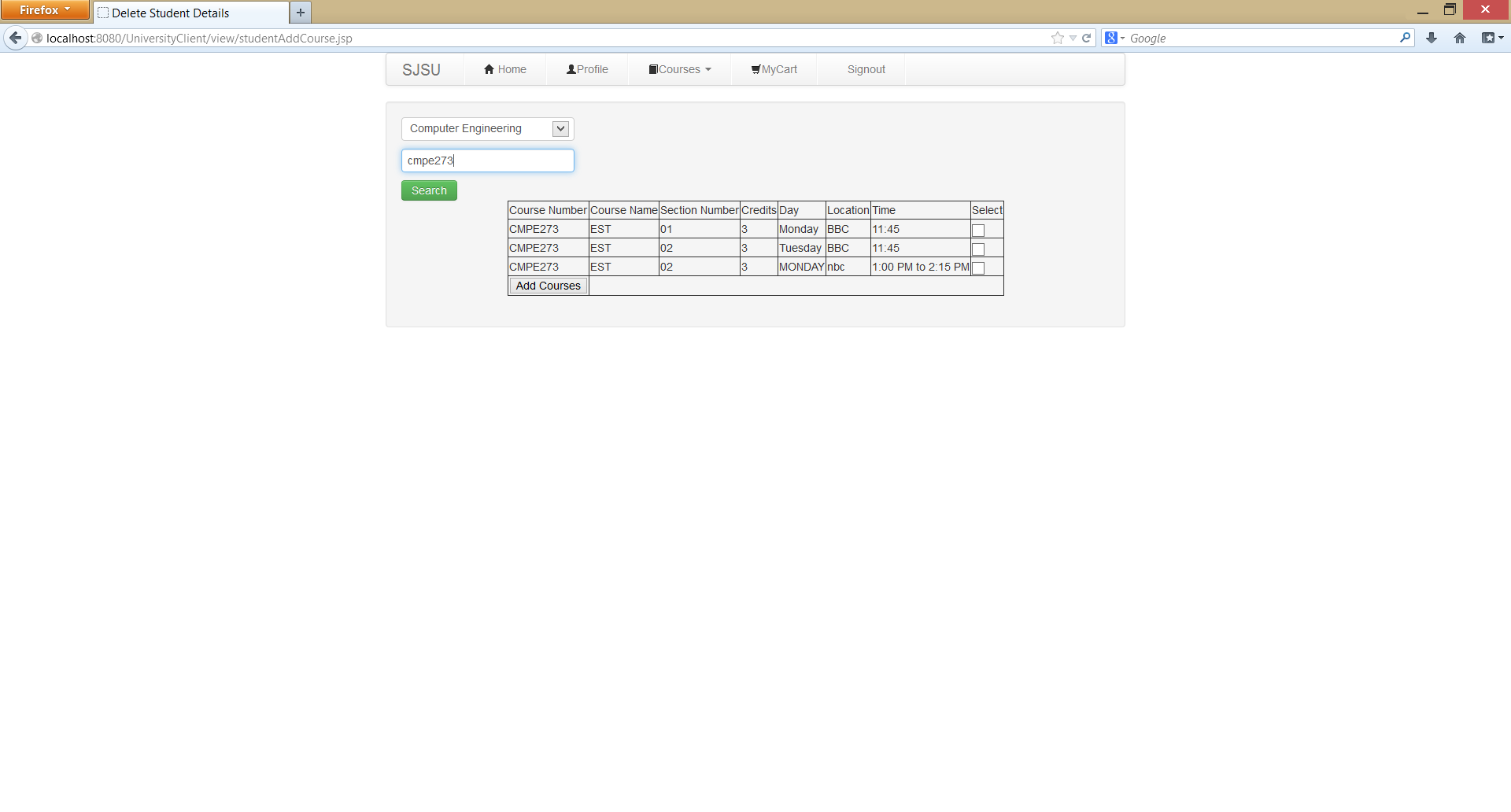
**Member Edit:**

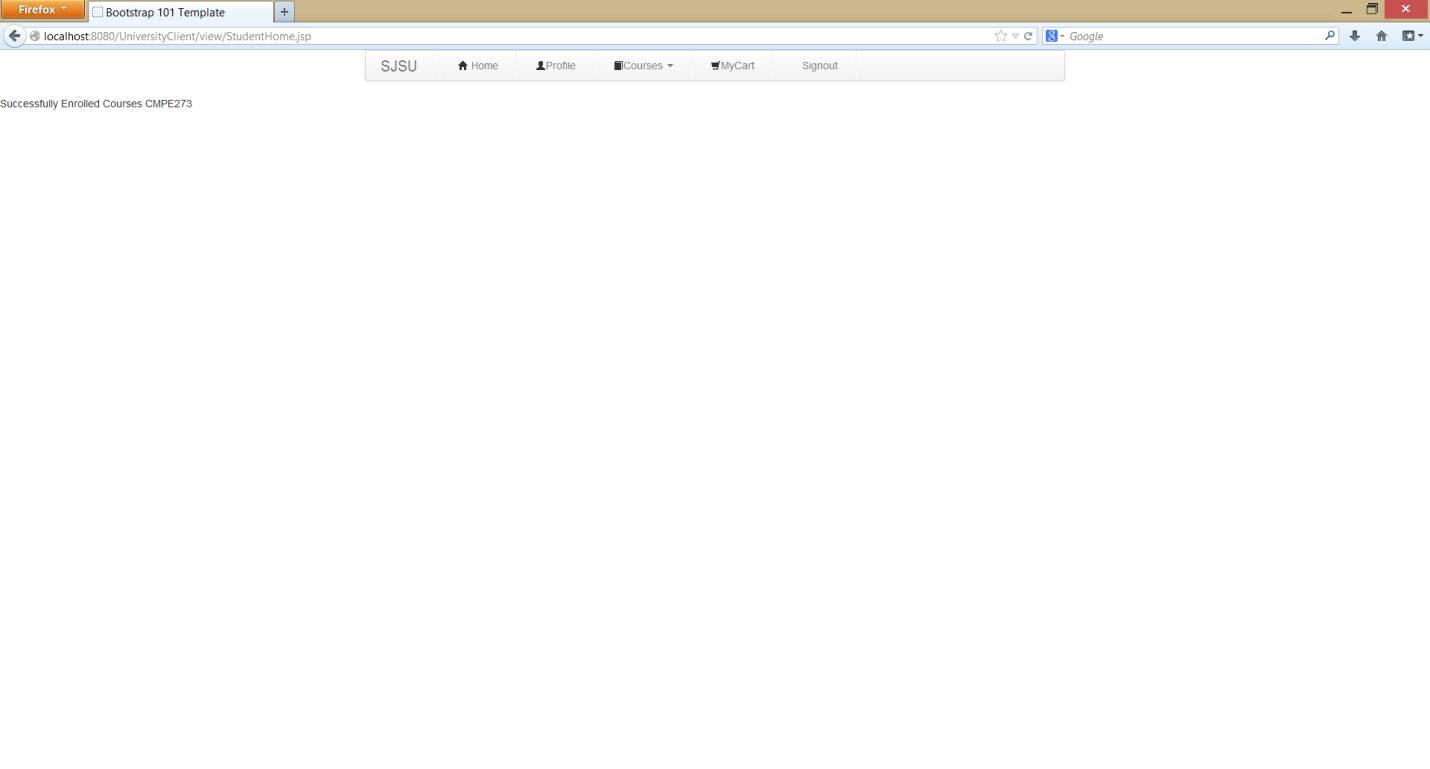
****

**Update Success:**

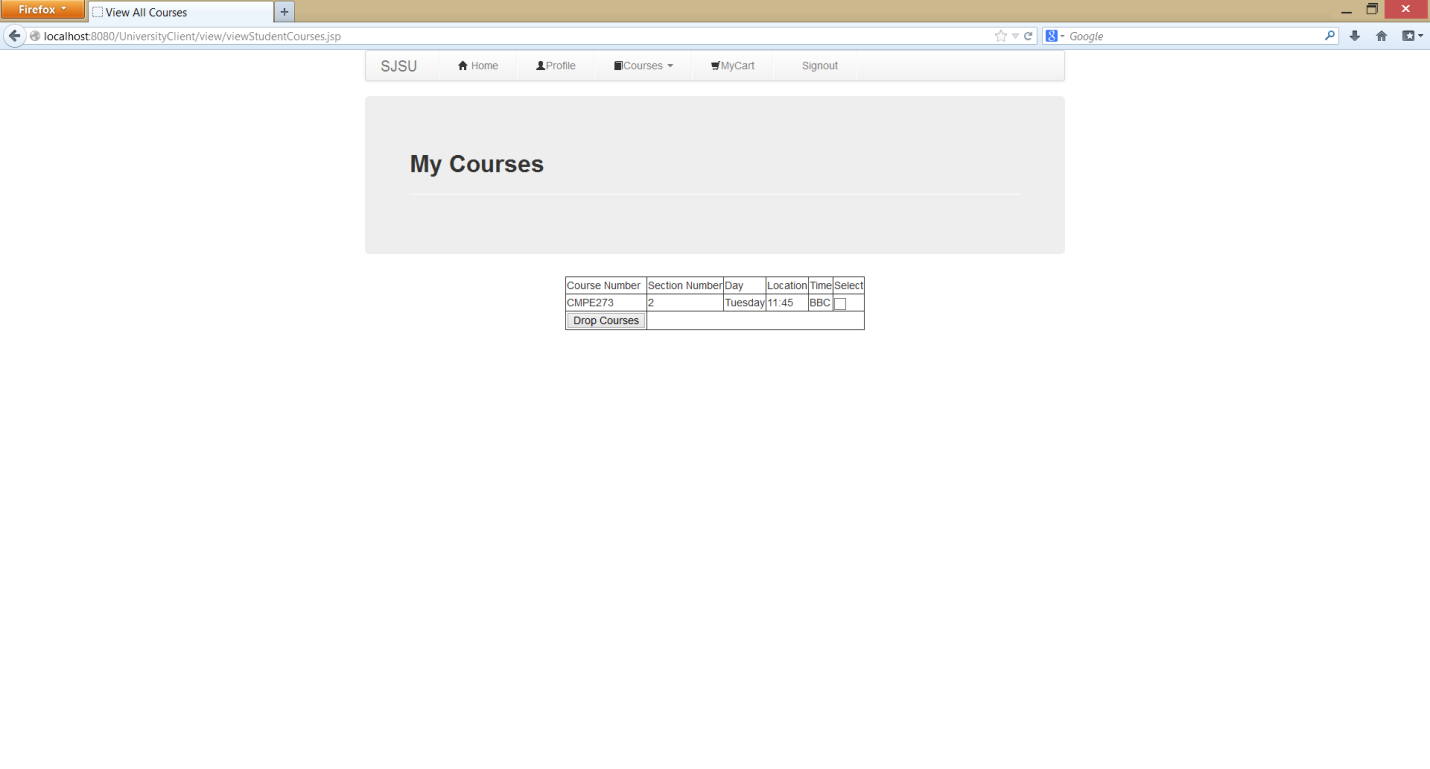
****

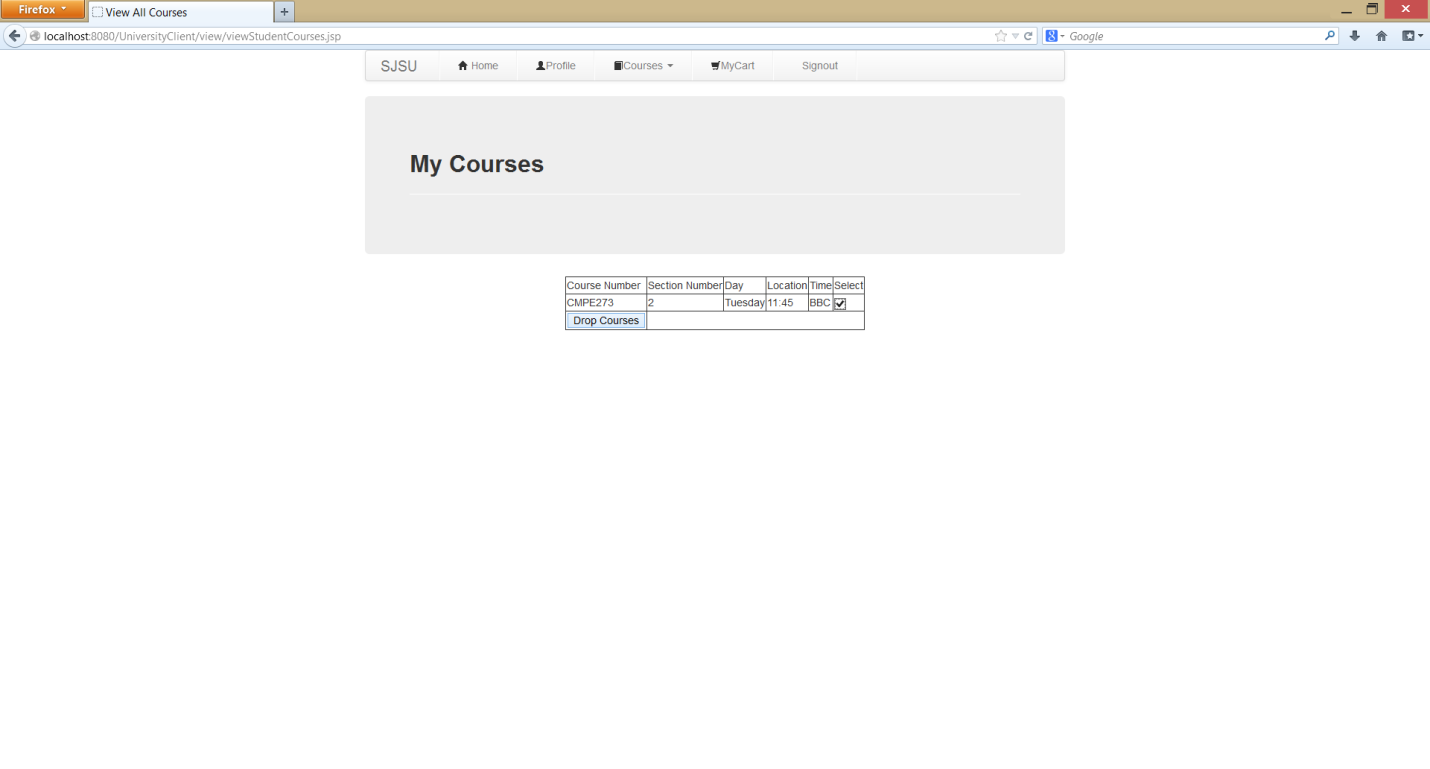
**Student Add Course:**

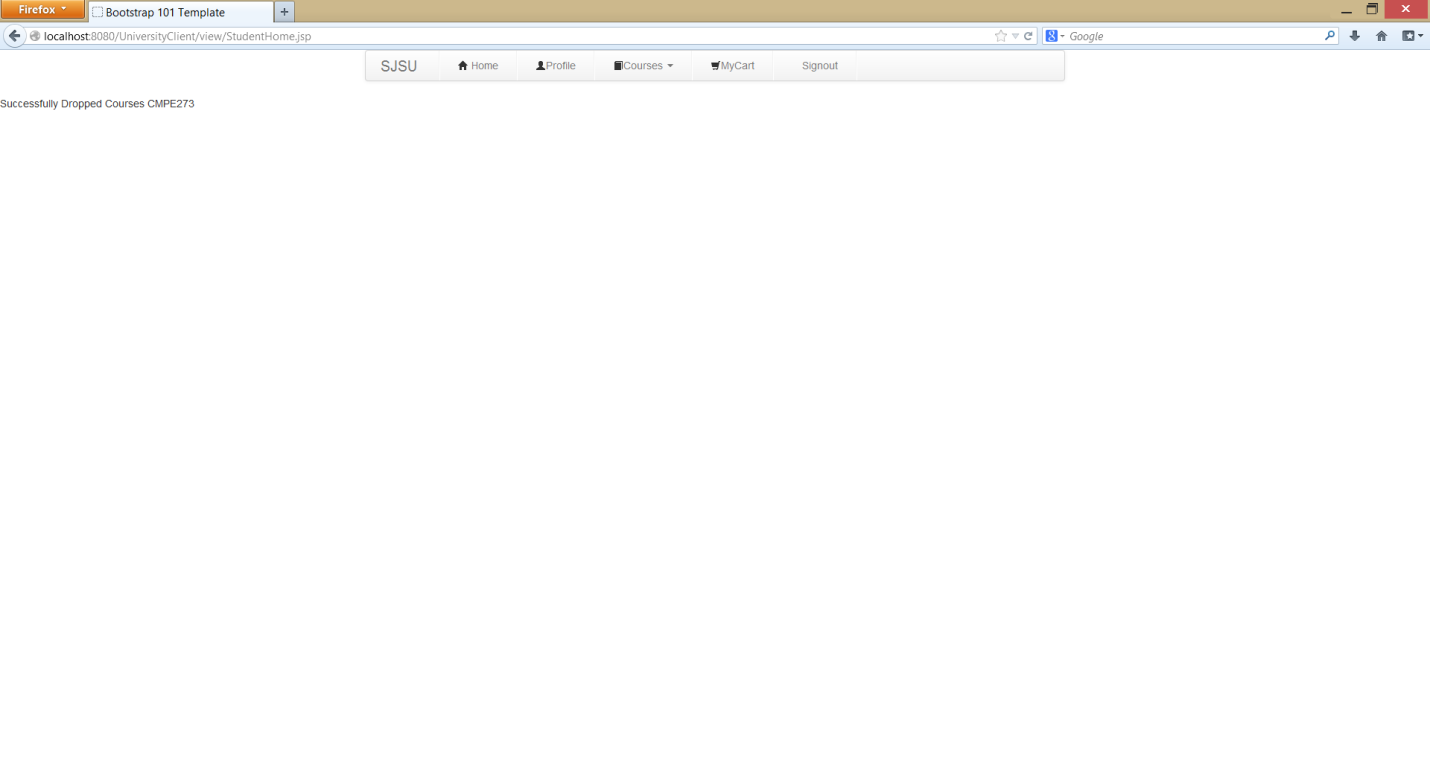
****

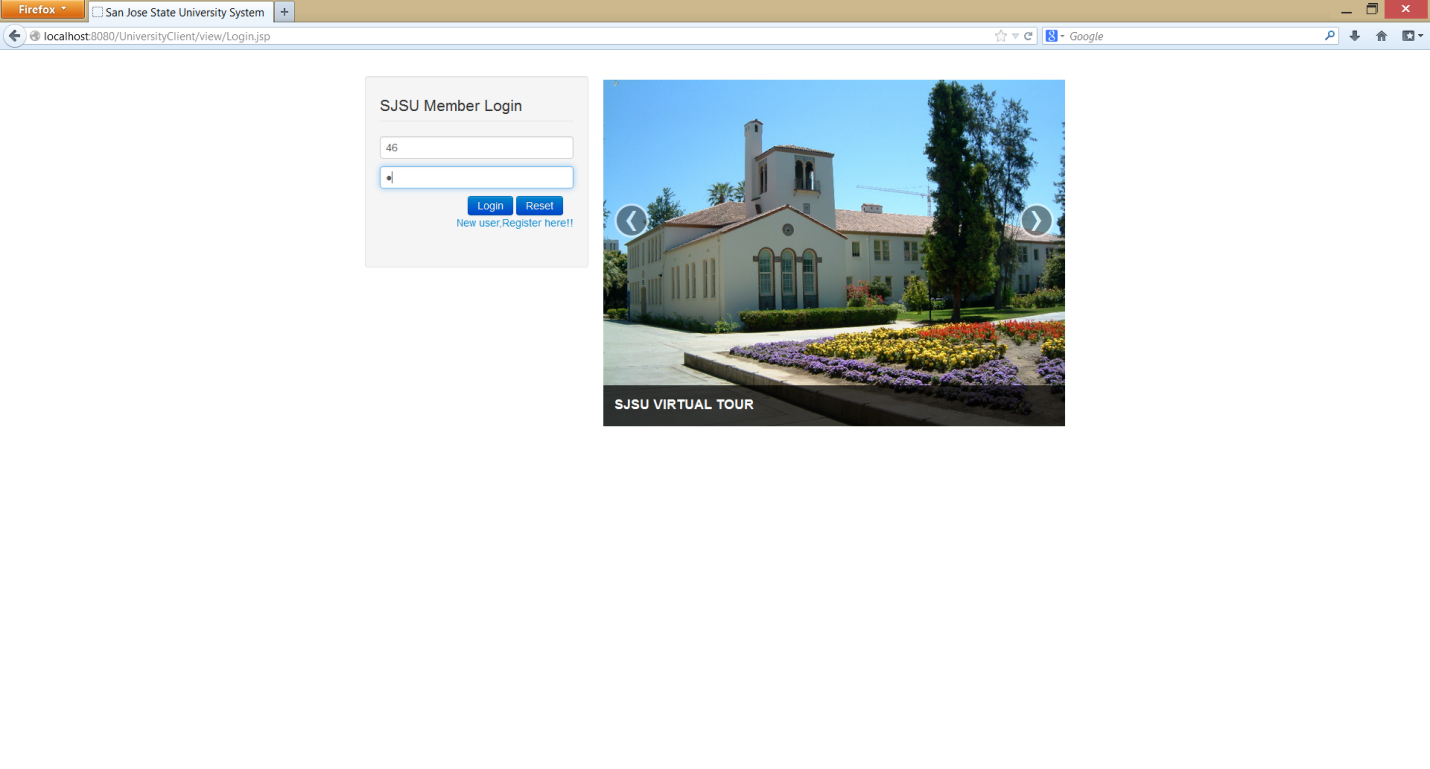
****

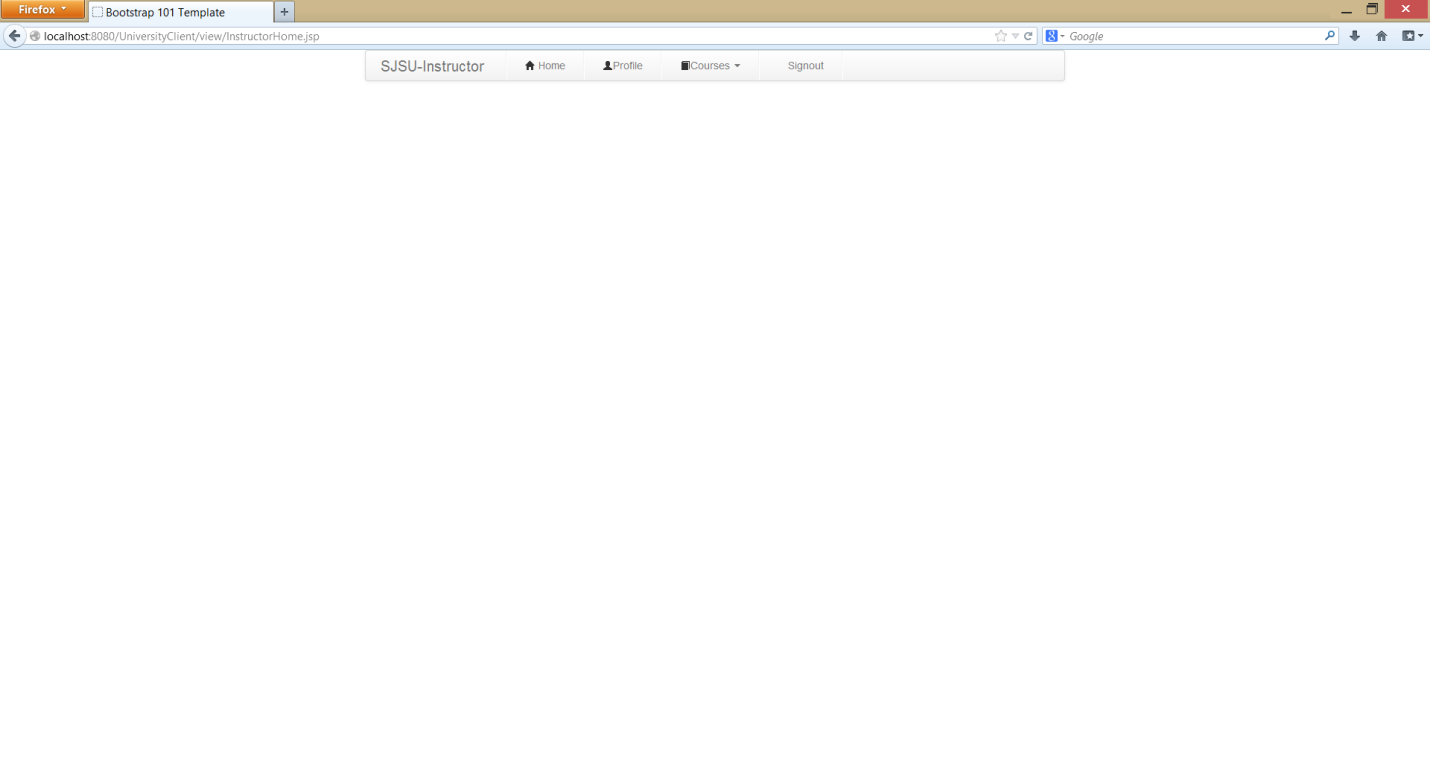
**View Enrolled Courses:**

****

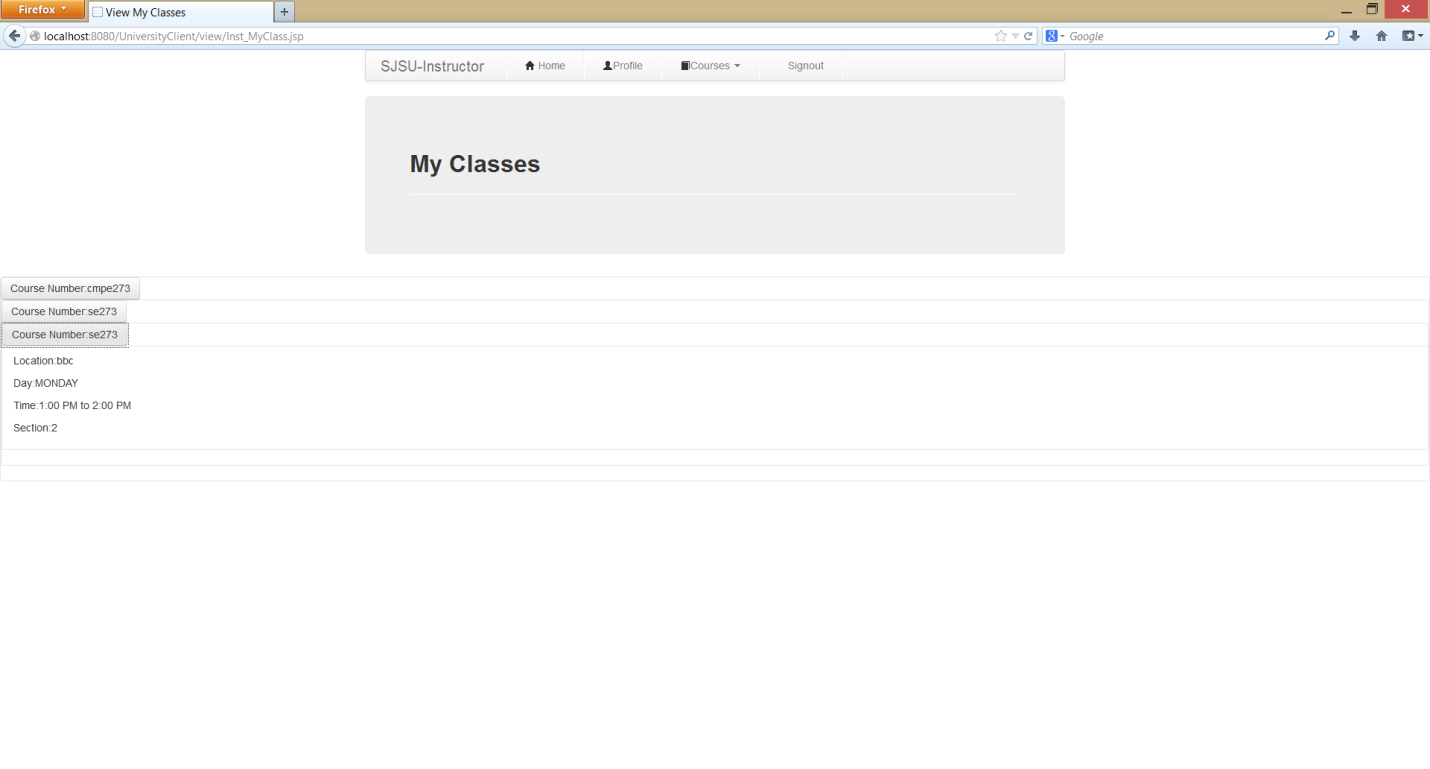
**Drop course:  
 **

****

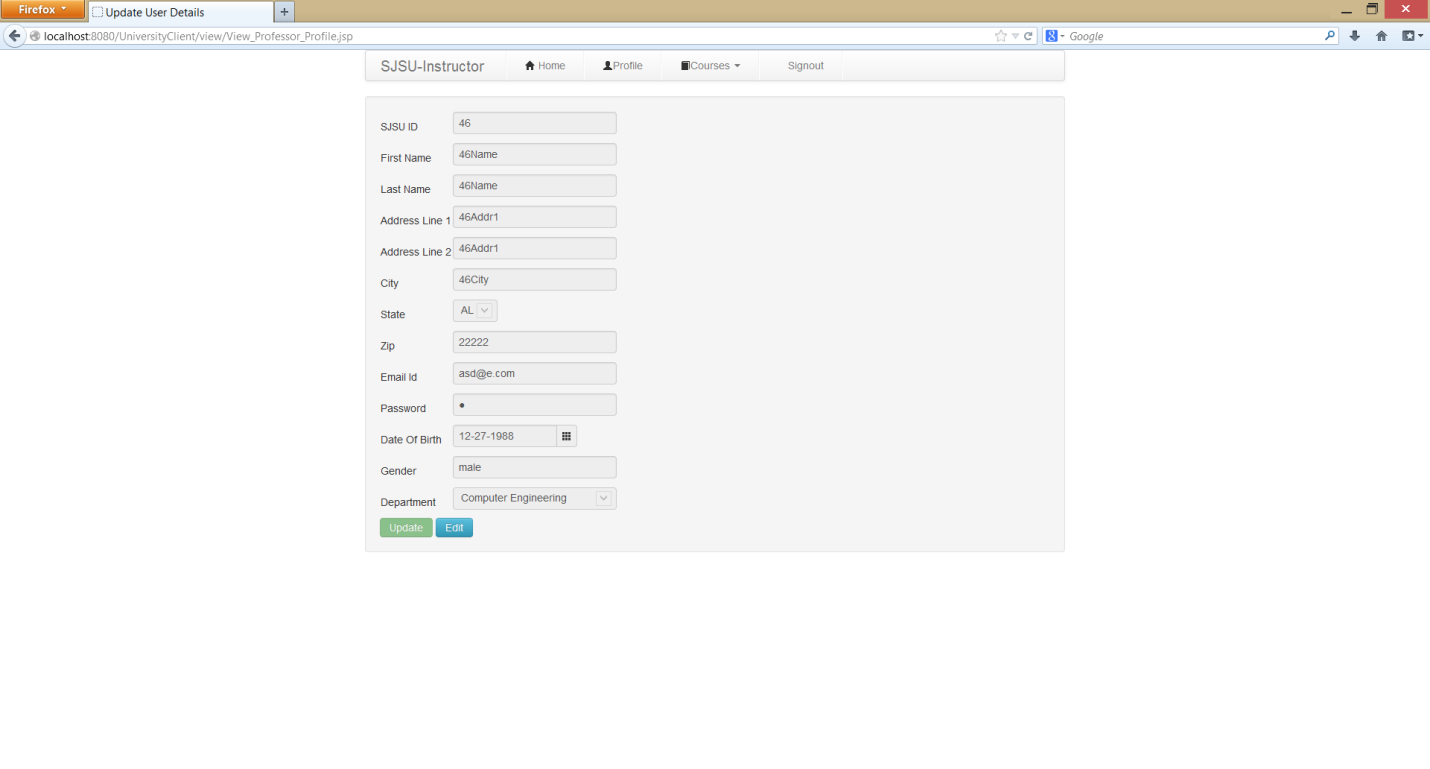
**Instructor Login:  
**

****

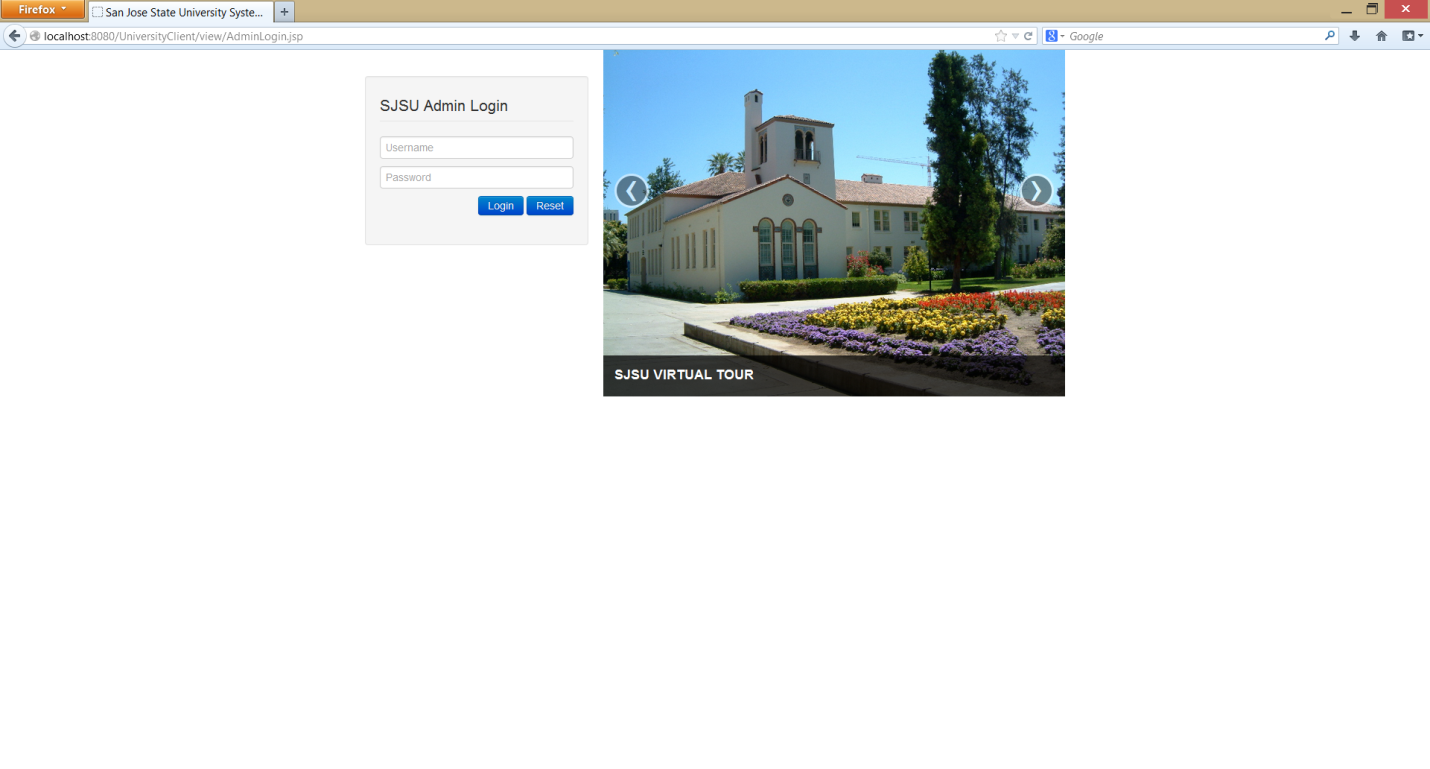
**Instructor Assigned Courses:**

****

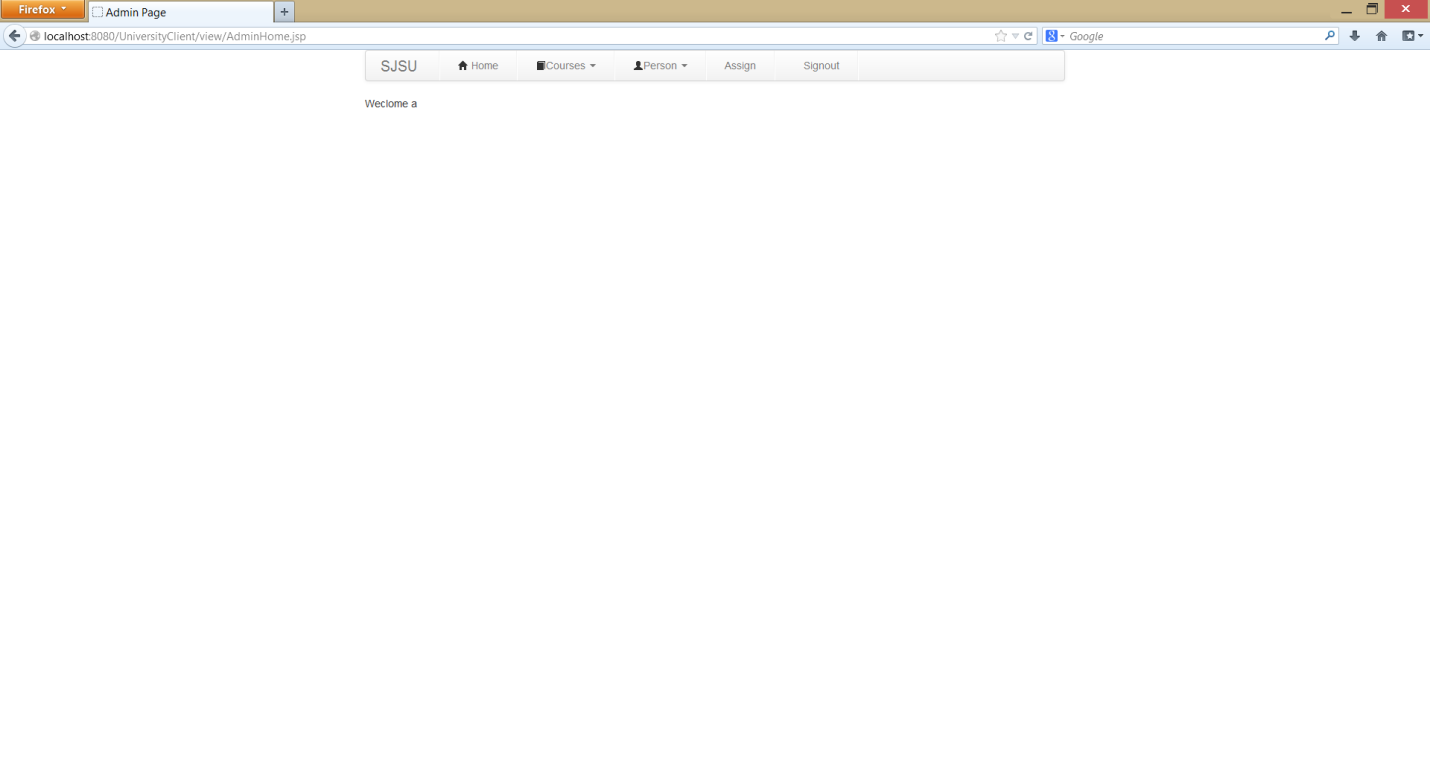
**Instructor view profile:**

****

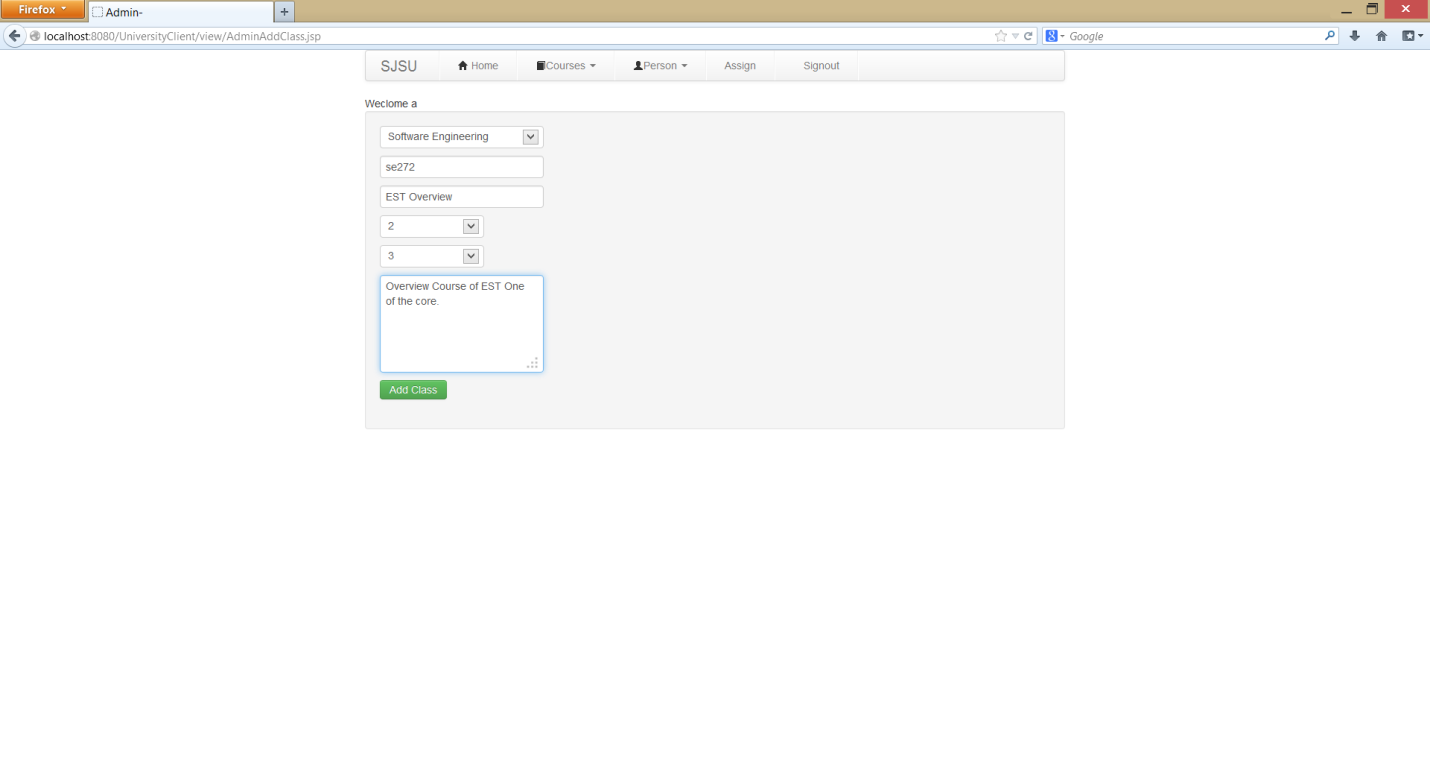
**Admin Login:**

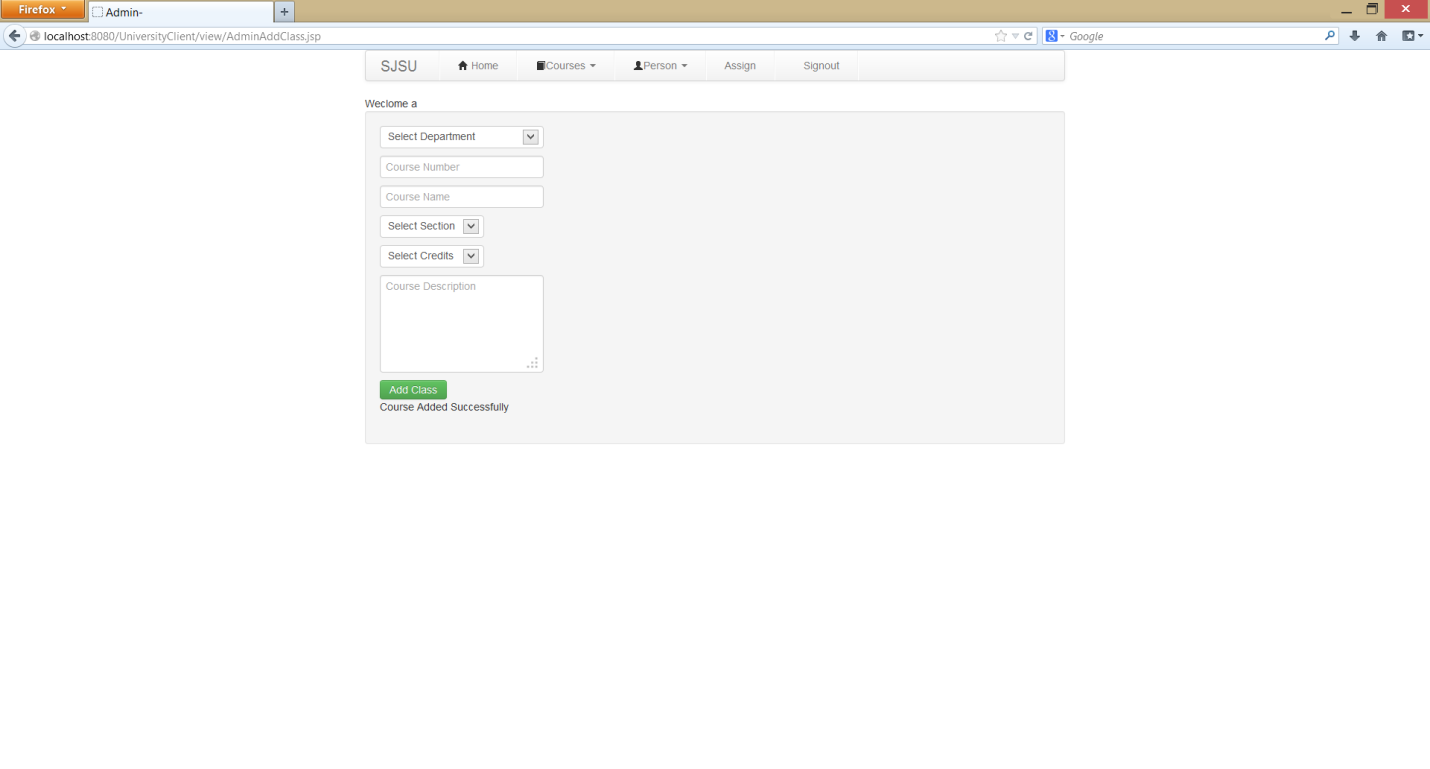
****

**Admin Home:**

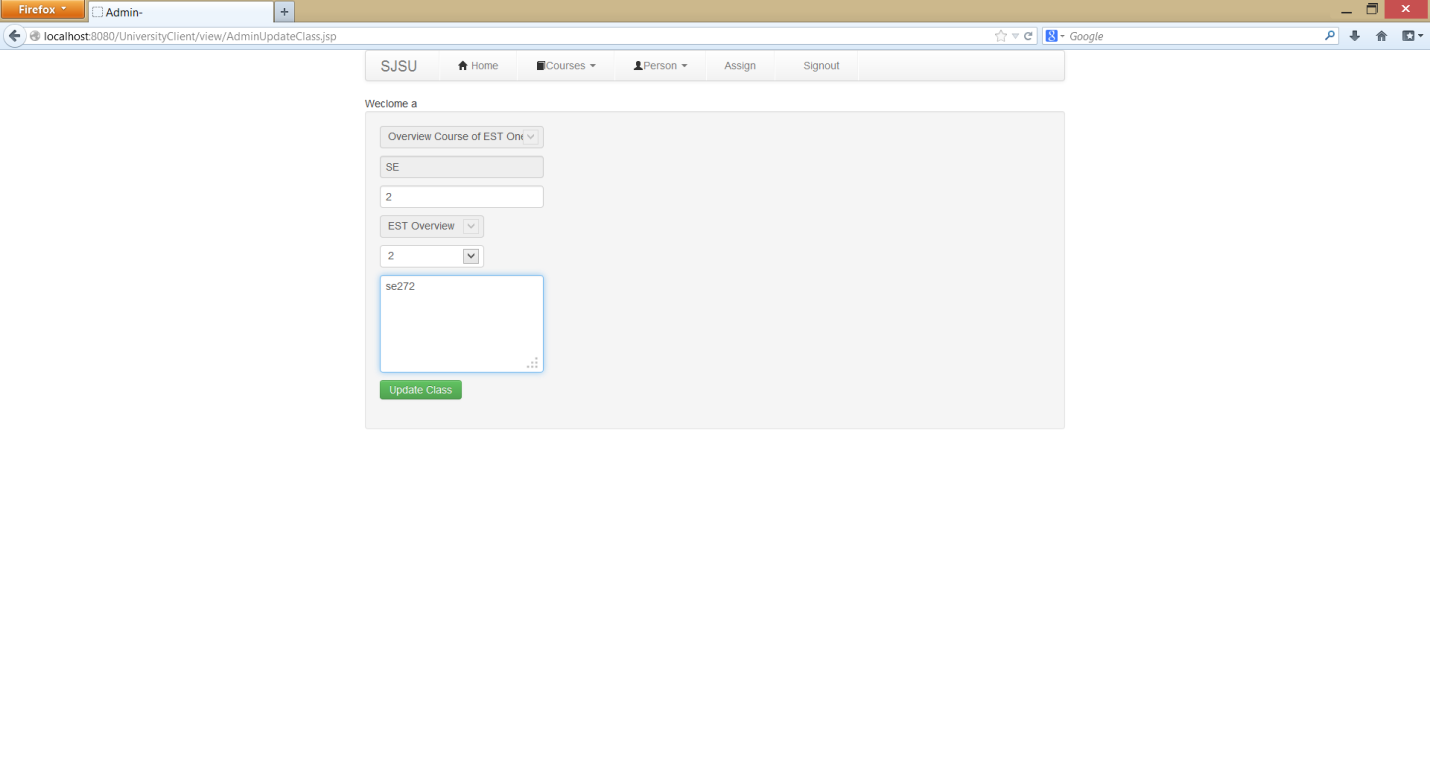
****

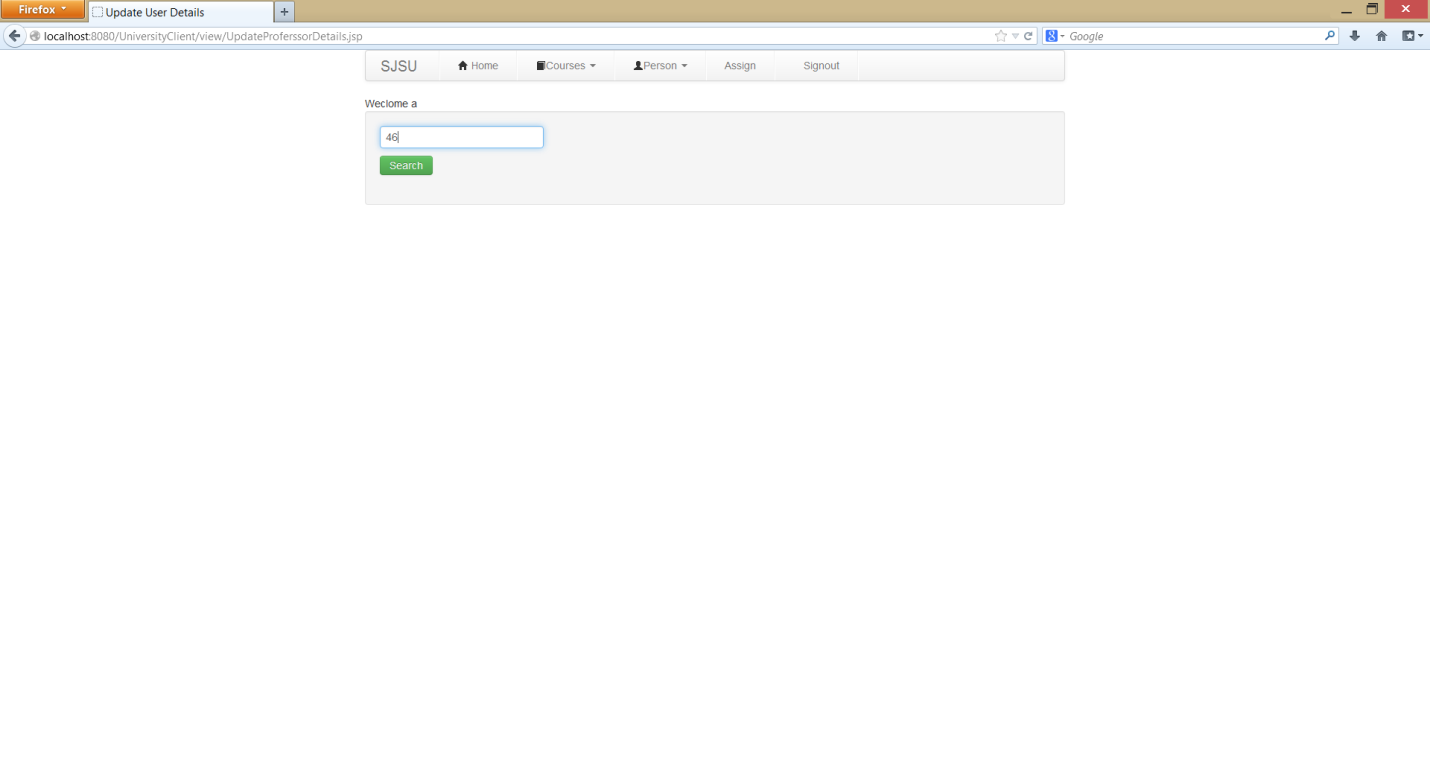
**Admin Add Course:**

****

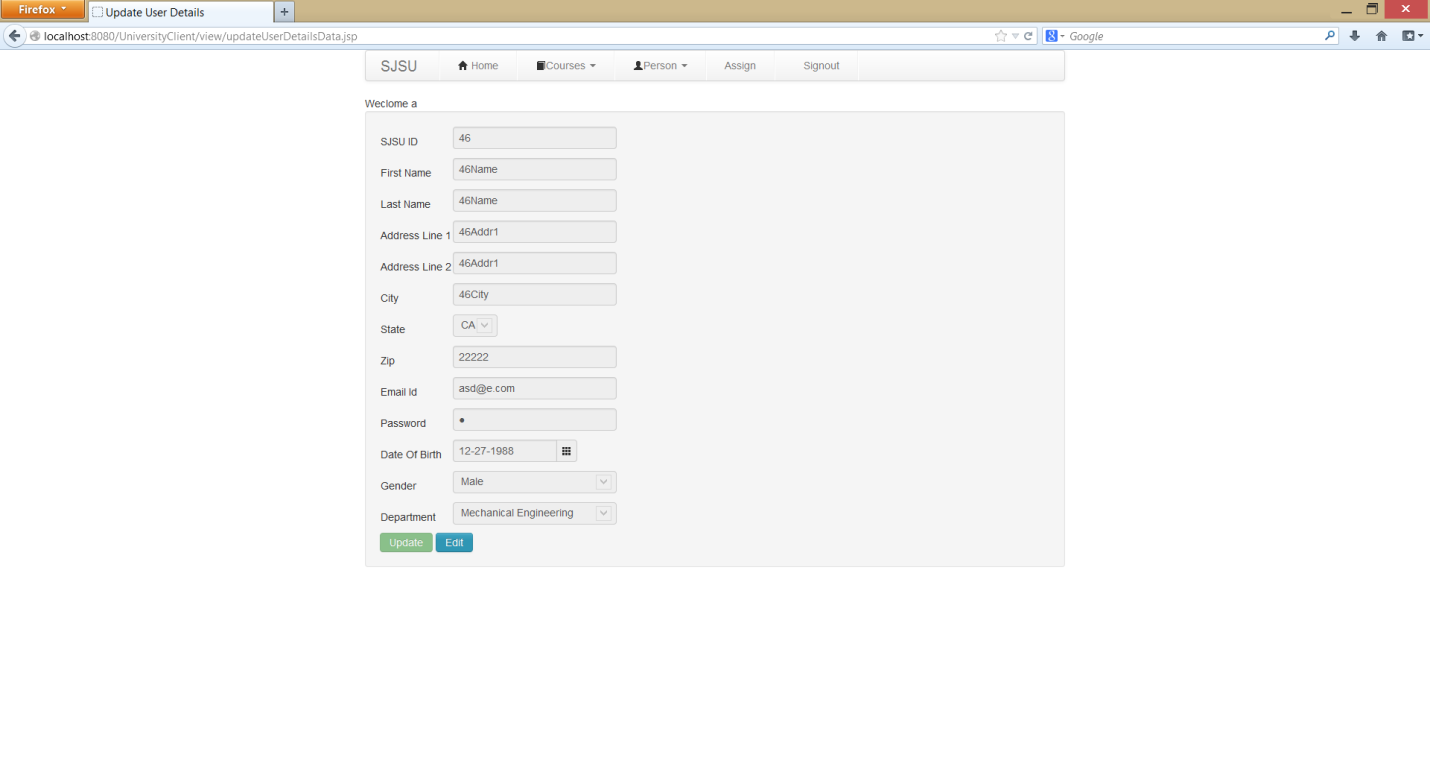
****

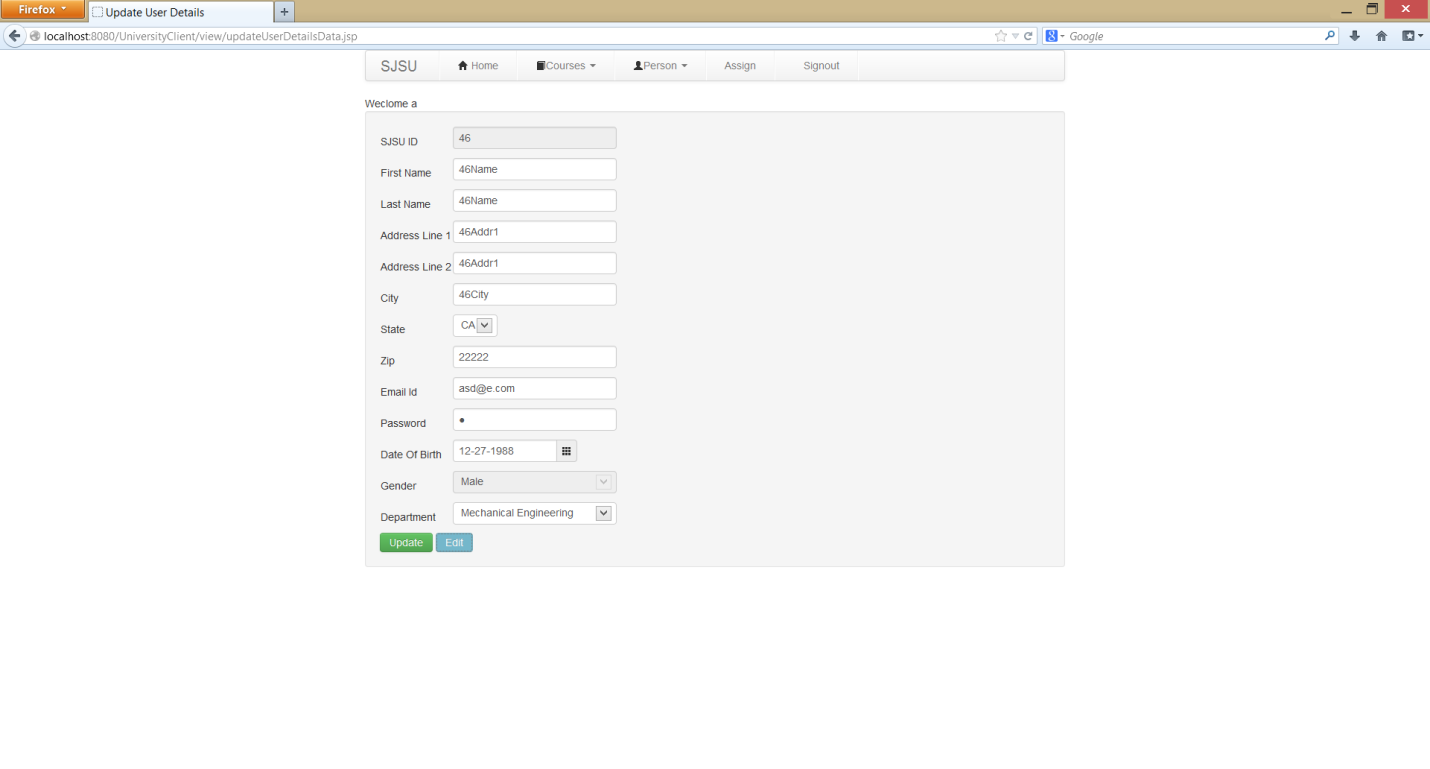
**View Course:**

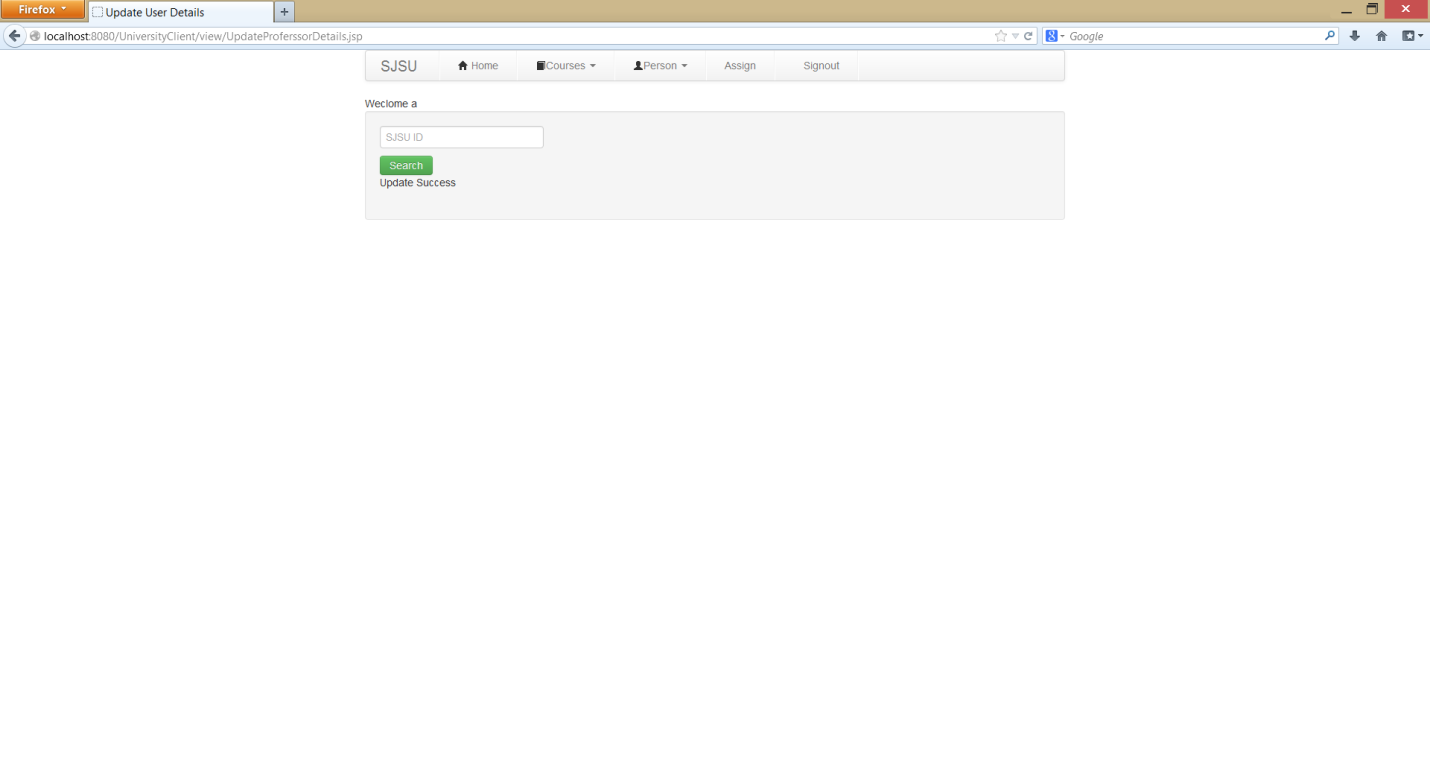
****

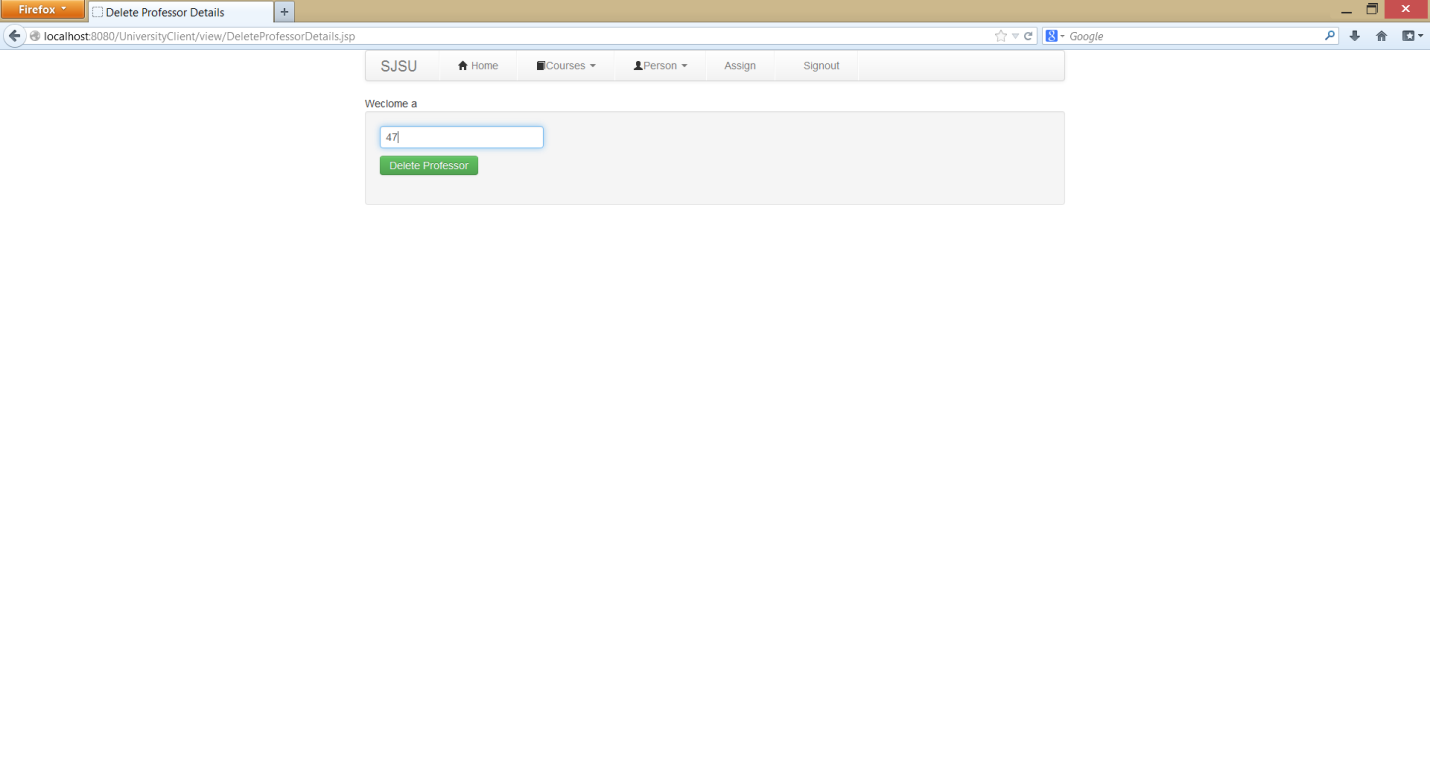
**Edit User Information:  
**

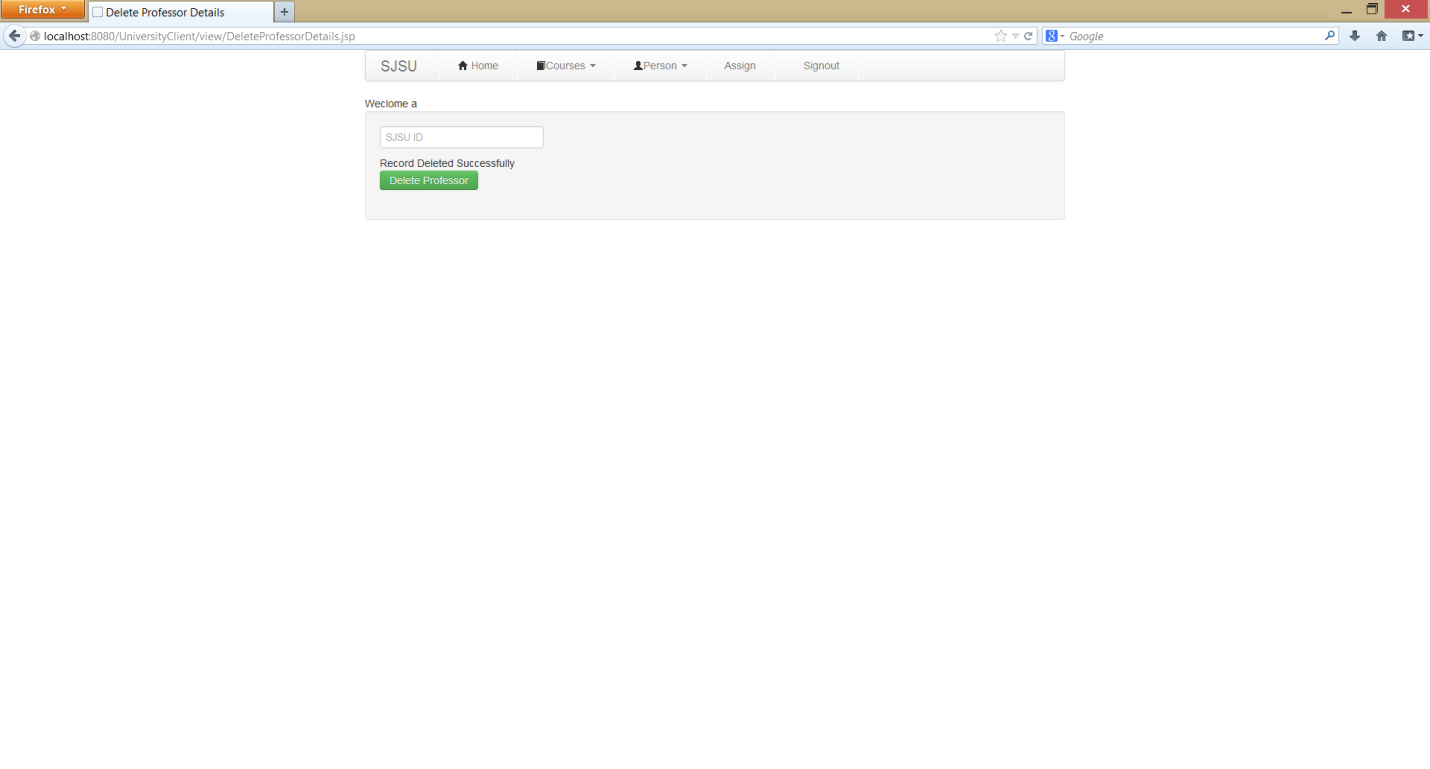
**View User Information:**

****

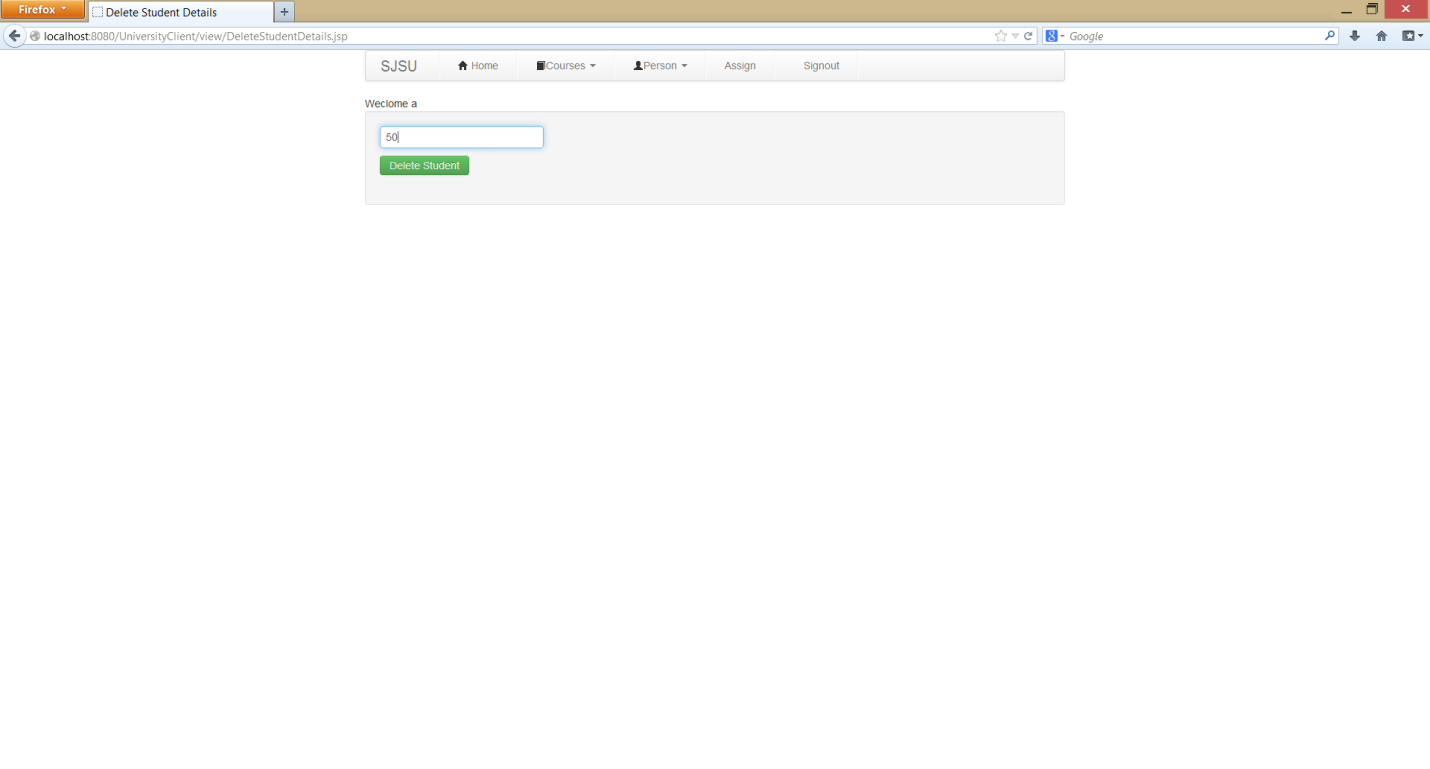
**Edit User Information  
**

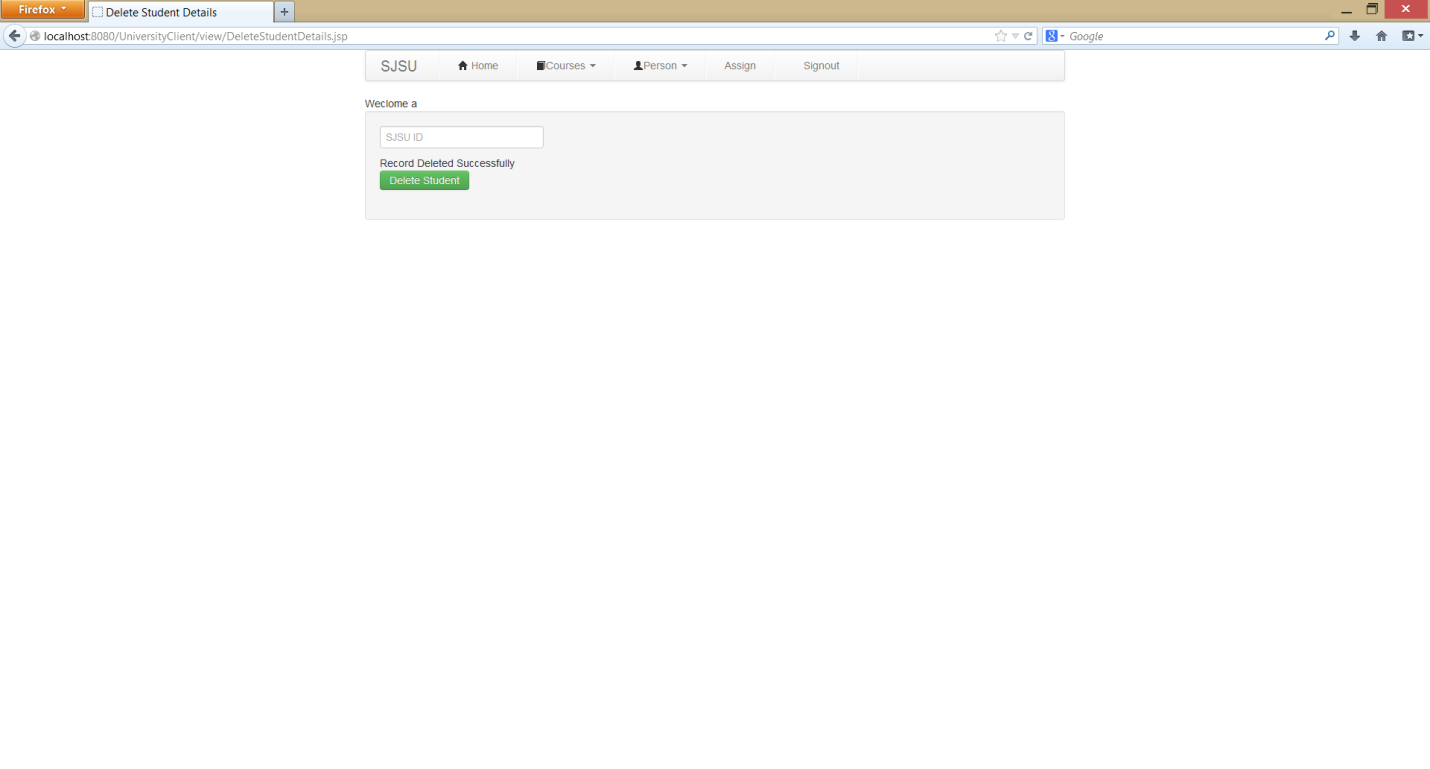
****

**Delete Professor:  
**

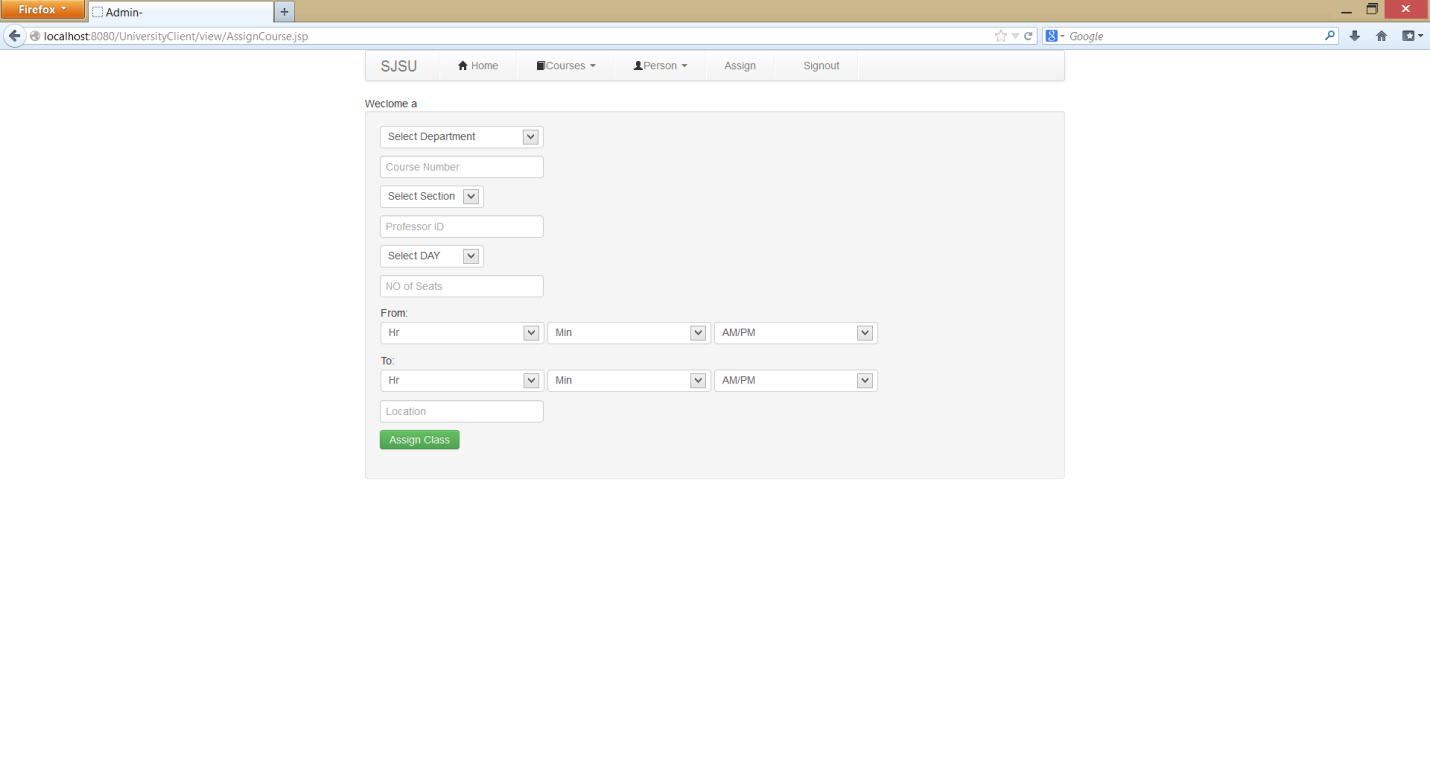
****

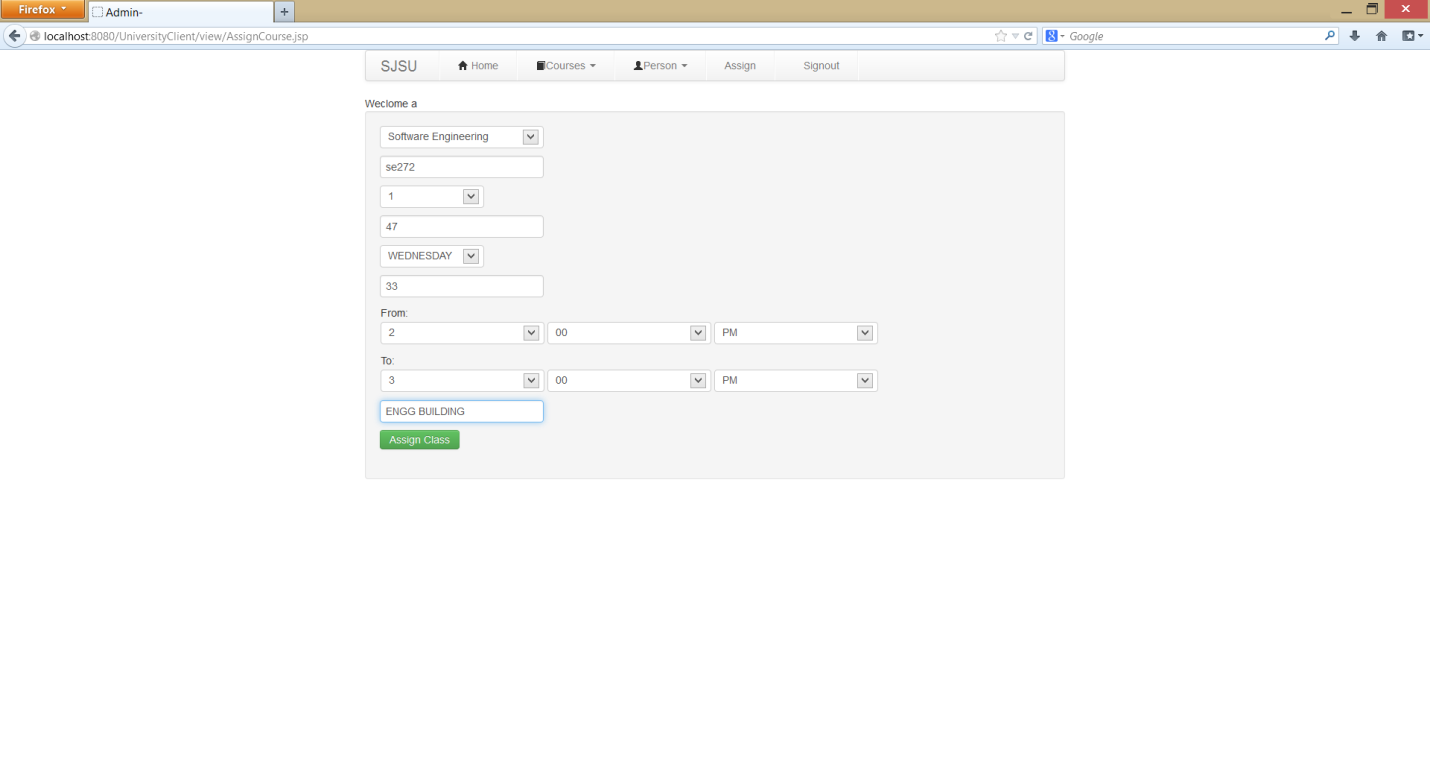
**Delete Student:**

****

****

**Admin Assign Course:**

****

****

**Observations and lessons learned**

* Scalability is very important for large systems. Testing with high load may show performance degradation. The system breaks with very heavy load. We have to handle heavy resources very carefully.
* Database connections are very expensive and we should try and reuse it and use as less as possible for this resources
* Connection pooling, batch processing and caching are good options for above.
* Database crashing can happen and is a possibility and we should leave the database in a consistent state. All operations should always be enclosed in transactions.
* We learned how to handle databases with huge loads.

**Conclusion :**

We have tried to build an enterprise Student Registration System. During the process we have learnt, how to take care of requirements like scalability, availability, reliability and accuracy.