**Airline Management System**

1. **Introduction**

The project “Airline Management System” comprises of a large number of flights which belong to a particular airline. The system we have implemented manages different objects viz.

* Airline
* Airline Employees
* Customers/Traveller

Each of these accesses a database schema which has corresponding tables. Web services are used for middleware technology.

1. **Individual Contributions**

|  |  |
| --- | --- |
| **Name** | **Contribution** |
| Name | * DB tables creation and modification – FlightDetails, Person * searching flights, update flights, display flight information, list all flights * Code Integration |
| Name | * DB tables creation and modification – Traveller, Person, Reservation, Reservation details * Reservation module – manage reservations, traveller, validations * Object caching implementation * Code Integration |
| Name | * DB tables creation and modification – Person, Employee * Employee module: Create Employee, Delete employee, Update employee info, List/Display employee info, Search employees * Project report preparation * Test Harness * Code Integration |
| Name | * DB tables creation and modification – Person, Payment * Database Connection Pooling – code implementation * JMS Implementation * Test Harness – Soap UI * Code Integration |
| Name | * DB tables creation and modification * Profiling module – login, create user, update user, delete user, list customers * Code Integration |

1. **Project Requirements**

**Tier 1 – Client Requirements**

* Airline management system client has all the functionalities listed as a part of requirements.
* A simple UI is used and appropriate error messages are displayed to the users wherever necessary. Some of us in the team have used a front end framework called Bootstrap which contains customizable style sheets and is a faster way to do client side coding.

**Tier 2 - Middleware**

* We have clearly defined interfaces implementing all the required functionalities.
* JDBC connection is established for select/insert/update data to database.
* JMS is used for publishing the status of flights to all the users who are online.

**Tier 3 – Database schema and database creation**

* My SQL is used for designing our database.

The following tables are created in the database,

Person, Employee, Traveller, Reservations, Reservation\_Details, FlightDetails, Payment

1. **Functional Requirements Implemented**

· Create a new employee

· Delete an existing employee

· Create a new Customer

· Delete an existing Customer

· Create a new reservation

· Cancel an existing reservation

· Issue a flight ticket

· Payment options

· List all customers known by the system

· List all employees known by the system

· List all reservation known by the system

· List all flights known by the system

· Change a employee/ customer information (name, address, etc) ability to change ALL attributes

· Change a flights information (time, source, destination etc) change ALL attributes

· Search for a employee based on attributes.

· Search for a flight based on attributes

· Display information about a employee

· Display information about a traveler

· Display information about a flight

1. **Non Functional Requirements - Scalability, Performance and Reliability**

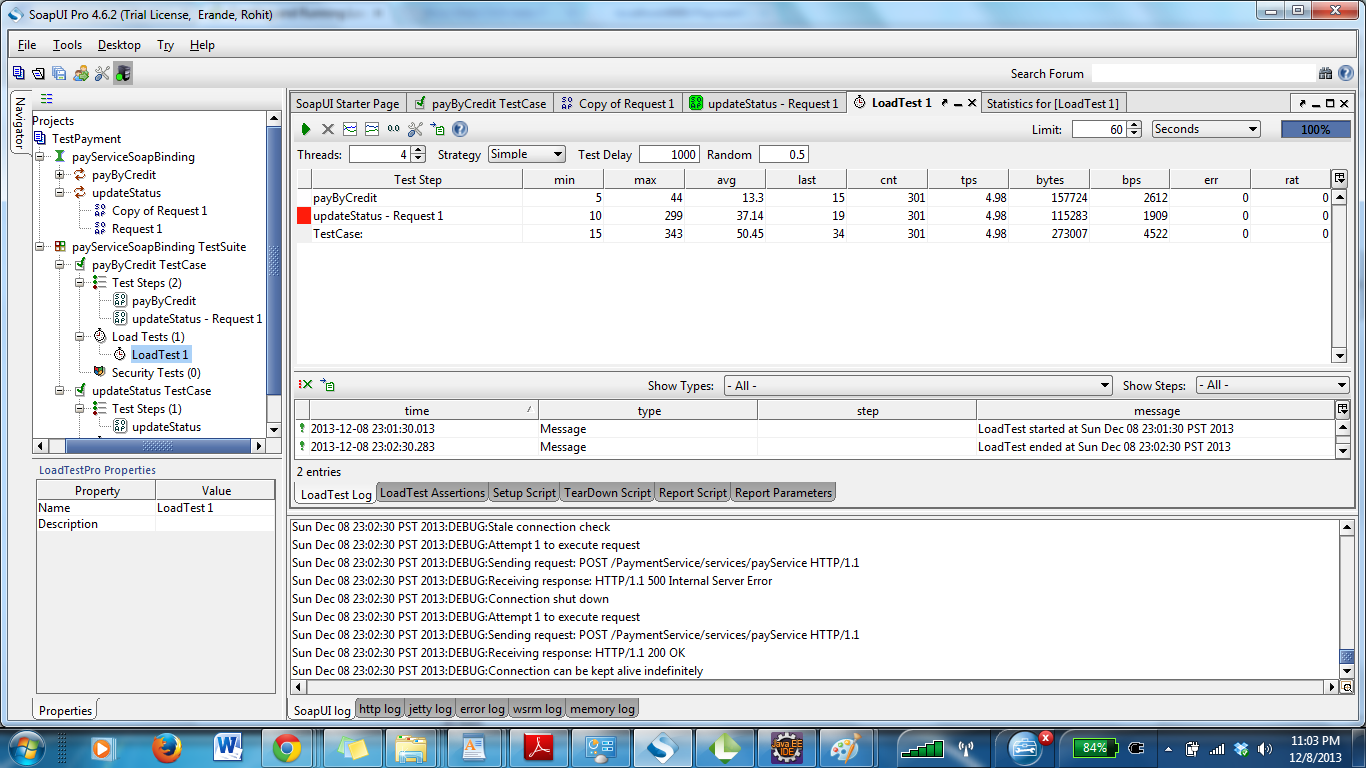
* Object caching is implemented for searching for flights(detailed explanation below)
* Database connection pooling is handled(detailed explanation below)
* The Person and Employee tables have 5000 records and we are successfully able to handle and show the data for such large number of records in our application.
* PreparedStatement is made use of in all the server methods to improve performance.
* SQL queries used to access the database are tuned. Some of the things we have focussed on include: Not using Like predicate in queries, using ‘=’ instead of IN when only one record has to be retrieved, using specific column names in select instead of \*.
* We have tested the flow of all the modules so that nothing crashes thus ensuring reliability

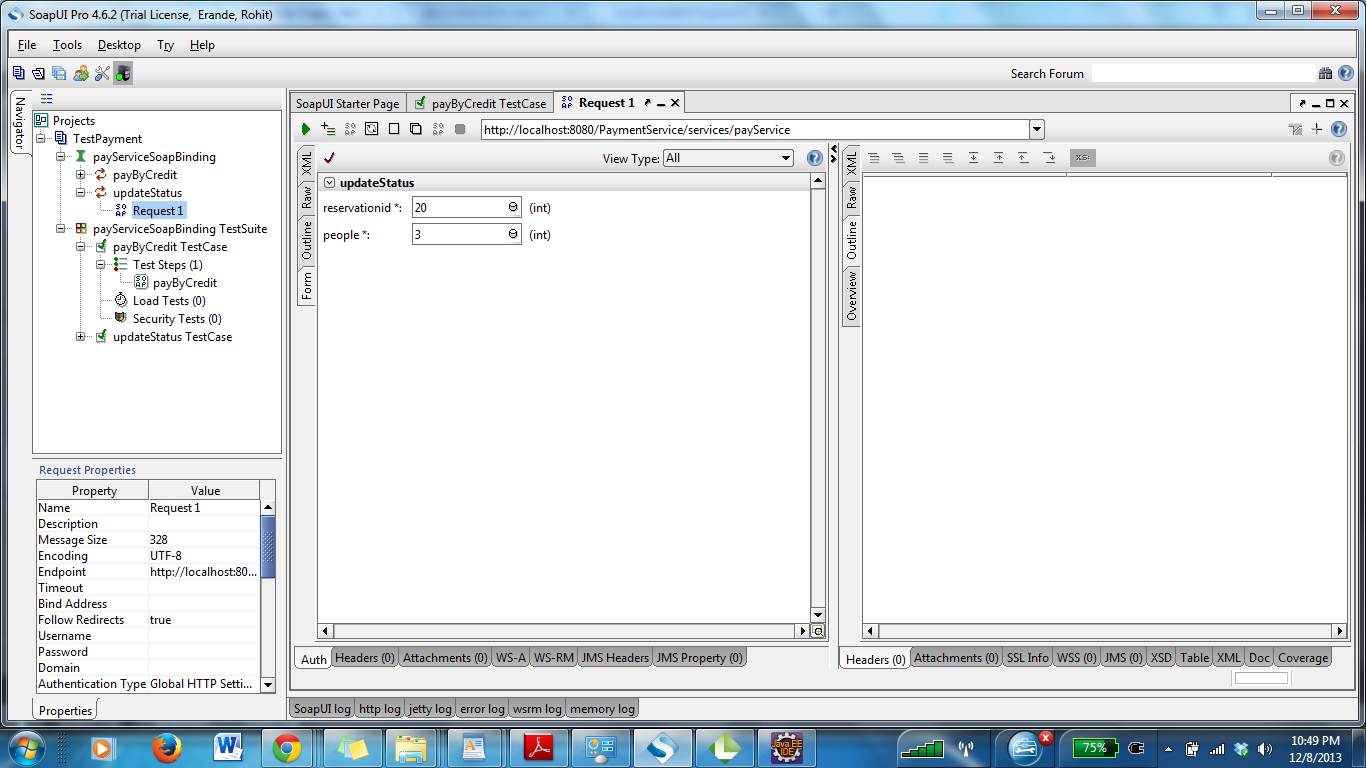
1. **Testing**

**Test Harness**

We have used **Soap UI** to test the performance of web services. When a new project is created, we give our wsdl and then add test steps for a specific method. We run the test against certain number of count.

The below screenshots show the test harness results for payment and update flight status,

****

****

1. **Object management policy**
2. **Object caching** is done for Searching for flights. This helps faster access as searching flights is one of the most frequently used functionalilities in an airline management system.

-We have used lazy loading for caching. On the first request for a flight search, the results are cached in a map with source and destination as key. The search results are mapped to bean files and are stored against the key.

-On the same request, the results are rendered from the cache thus saving database hits.

-When a flight is updated, the map is looked upon and the object in the map is updated accordingly.

The code listing of the object caching done is as shown below,

package servlets;

import java.util.Arrays;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import fdpackage.FlightDetailsBean;

public class FlightCaching {

public static Map<String, FlightDetailsBean[]> journeyToFlightsMap = new HashMap<String, FlightDetailsBean[]>();

public static void updateCache(String oldKey, String newKey, FlightDetailsBean updatedBean) {

if(journeyToFlightsMap!=null && journeyToFlightsMap.size()>0) {

if(journeyToFlightsMap.containsKey(oldKey) && journeyToFlightsMap.get(oldKey)!=null) {

List<FlightDetailsBean> flightList = Arrays.asList(journeyToFlightsMap.get(oldKey));

for(int i=0; i<flightList.size(); i++) {

if(flightList.get(i).getFlightId()==updatedBean.getFlightId()) {

if(newKey.equals(oldKey)) {

flightList.add(i, updatedBean);

} else {

flightList.remove(i);

}

}

}

journeyToFlightsMap.put(oldKey, (FlightDetailsBean[])flightList.toArray());

}

if(!oldKey.equals(newKey)) {

if(journeyToFlightsMap.containsKey(newKey) && journeyToFlightsMap.get(newKey)!=null) {

List<FlightDetailsBean> flightList = Arrays.asList(journeyToFlightsMap.get(newKey));

flightList.add(updatedBean);

journeyToFlightsMap.put(newKey, (FlightDetailsBean[])flightList.toArray());

}

}

}

}

public static FlightDetailsBean[] fetchDataFromCache(String key) {

System.out.println("Fetching data from cache for key : "+key);

if(journeyToFlightsMap!=null && journeyToFlightsMap.size()>0) {

if(journeyToFlightsMap.containsKey(key)) {

return journeyToFlightsMap.get(key);

} else {

return null;

}

} else

return null;

}

public static void storeDataInCache(String key, FlightDetailsBean[] flightBeans) {

System.out.println("Storing data from cache : "+key);

journeyToFlightsMap.put(key, flightBeans);

}

public static Map<String, FlightDetailsBean[]> getJourneyToFlightsMap() {

return journeyToFlightsMap;

}

public static void setJourneyToFlightsMap(

Map<String, FlightDetailsBean[]> journeyToFlightsMap) {

FlightCaching.journeyToFlightsMap = journeyToFlightsMap;

}

}

1. Java beans: We have used Java Bean classes to handle the functionality and the data entered. Beans have getter and setter methods for easy management.
2. **How heavyweight resources are handled**
3. **A pool of database connections**

Connection pooling is done to cache database connections which can be used when the database needs to be accessed. We make use of 2 java classes viz,

MaxPoolReachedException

ConnectionPool

The class DBHelper is used to access the instances and methods of ‘ConnectionPool’ and ‘MaxPoolReachedException’. Every module gets connections through DBHelper.java

The class ‘ConnectionPool’ maintains 10 connections. It has 2 methods

* public synchronized static Connection getConnection(String url,String username,String password)

This creates a new connection till the max limit i.e. 20 in this case is reached

* public synchronized static void addConnectionToPool(Connection con)

This method adds the connections created to a pool of connections

The class MaxPoolReachedException is a customized exception class for handling the exception when the number of connections exceeds the maximum pool size chosen (i.e. 20)

After the request is processed, the connections are added back to the connection pool and the next request can use the same connection from the pool.

1. **Object caching** as explained in the previous section (Object management policy)
2. Heavy weight resources like images are not used in our system.

**Connection pooling code**

package helperClasses;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

public class DBHelper {

public static Connection getConnection() throws Exception {

Connection conn = null;

try{

Class.forName("com.mysql.jdbc.Driver").newInstance();

/\*conn = ConnectionPool.getConnection("jdbc:mysql://localhost:3306/airline","root","root");\*/

conn = ConnectionPool.getConnection("jdbc:mysql://localhost:3306/airline","root","root");

} catch(Exception e) {

System.out.println("Exception in connecting to DB : ");

e.printStackTrace();

throw e;

}

return conn;

}

public static ResultSet getResults(String query) throws Exception{

Connection conn = getConnection();

ResultSet results = null;

if(!conn.isClosed()) {

Statement stmt = conn.createStatement();

results = stmt.executeQuery(query);

}

return results;

}

public static int executeQuery(String query) throws Exception{

Connection conn = getConnection();

int updatedRows = 0;

if(!connectionPool.addtoConnectionPool(conn)) {

Statement stmt = conn.createStatement();

updatedRows = stmt.executeUpdate(query);

}

return updatedRows;

}

}

package helperClasses;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.util.Vector;

public class ConnectionPool {

private static final int MAX\_POOL\_SIZE=10;

private static int CURRENT\_POOL\_SIZE;

public static Vector<Connection> connectionList=new Vector<Connection>(MAX\_POOL\_SIZE);

public synchronized static Connection getConnection(String url,String username,String password) throws

SQLException

{

    if(connectionList.isEmpty())

    {

    if(CURRENT\_POOL\_SIZE>=MAX\_POOL\_SIZE)

    {

    throw new MaxPoolReachedException("Maximum pool size reached cannot create connections");

    }

    CURRENT\_POOL\_SIZE++;

    System.out.println("creating new connection");

return DriverManager.getConnection(url, username,password);

    }

    else

    {

    CURRENT\_POOL\_SIZE--;

    System.out.println("returning from existing connection pool");

    return connectionList.get(0);

    }

}

public synchronized static void addConnectionToPool(Connection con){

System.out.println("adding back to connection pool");

connectionList.add(con);

}

}

package helperClasses;

public class MaxPoolReachedException extends RuntimeException

{

public MaxPoolReachedException(String message)

{

super(message);

}

}

1. **Policy used for deciding when to write data into the database**

Data entered from the UI which

-Doesn’t change often

-Need to available for retrieval

are written into the database.

For instance, customer information once entered may not be changed frequently although small changes like updating email ID, Address etc can be done. So, such information will be stored in the database as caching not frequently changing data is not good from performance perspective.

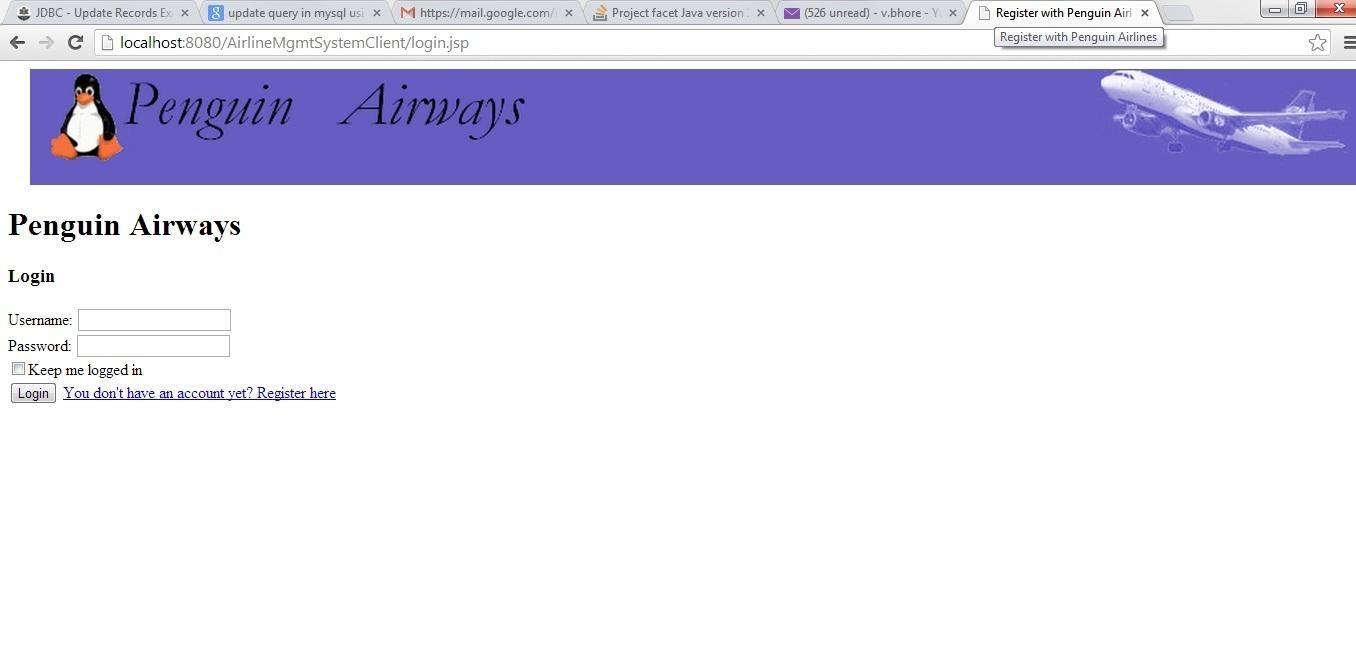
Example: Details of a Person which can be Employee and Traveller data are written to the database as it is composed of common data across Traveller and Exmployee and need to be retrieved upon querying for details related to both.

Object caching is implemented for searching flight information. This saves database hits for frequently accessed data.

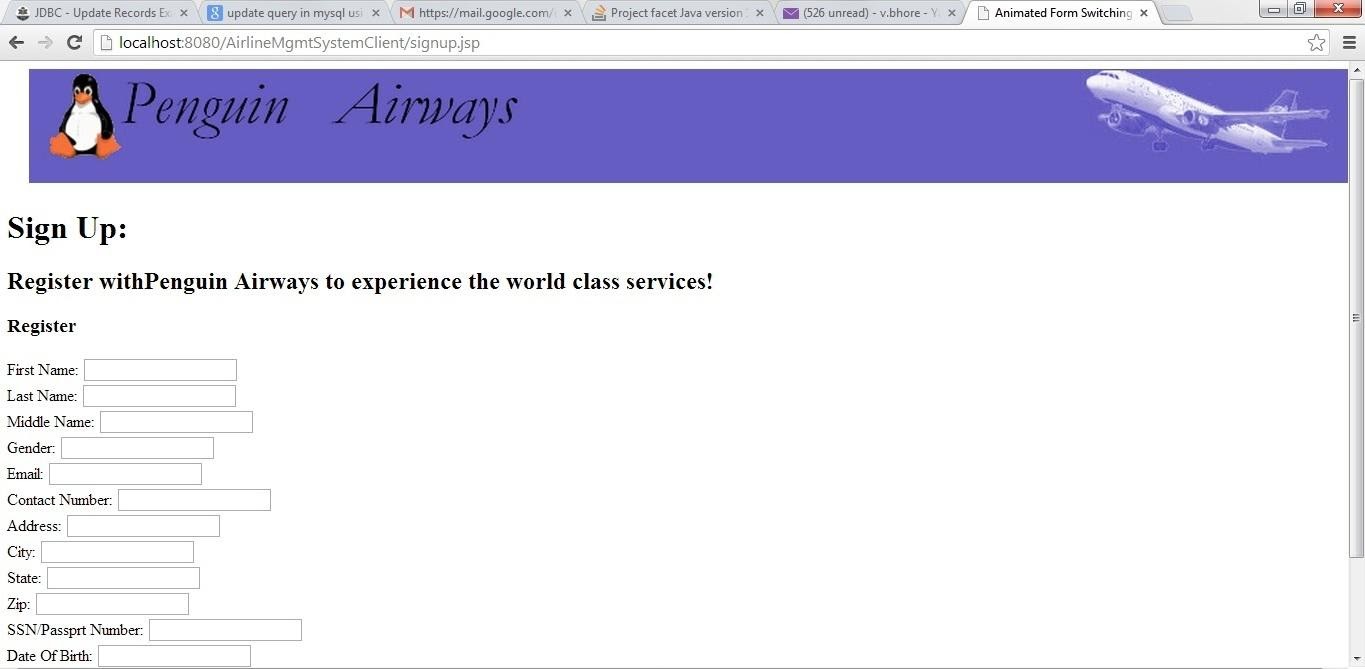
So, as per our policy, we have used lazy loading for caching. On the first request for a flight search, the results are cached in a map with source and destination as key. The search results are mapped to bean files and are stored against the key.When a flight is updated, the map is looked upon and the object in the map is updated accordingly.

1. **Screen Capture of Client Applications**

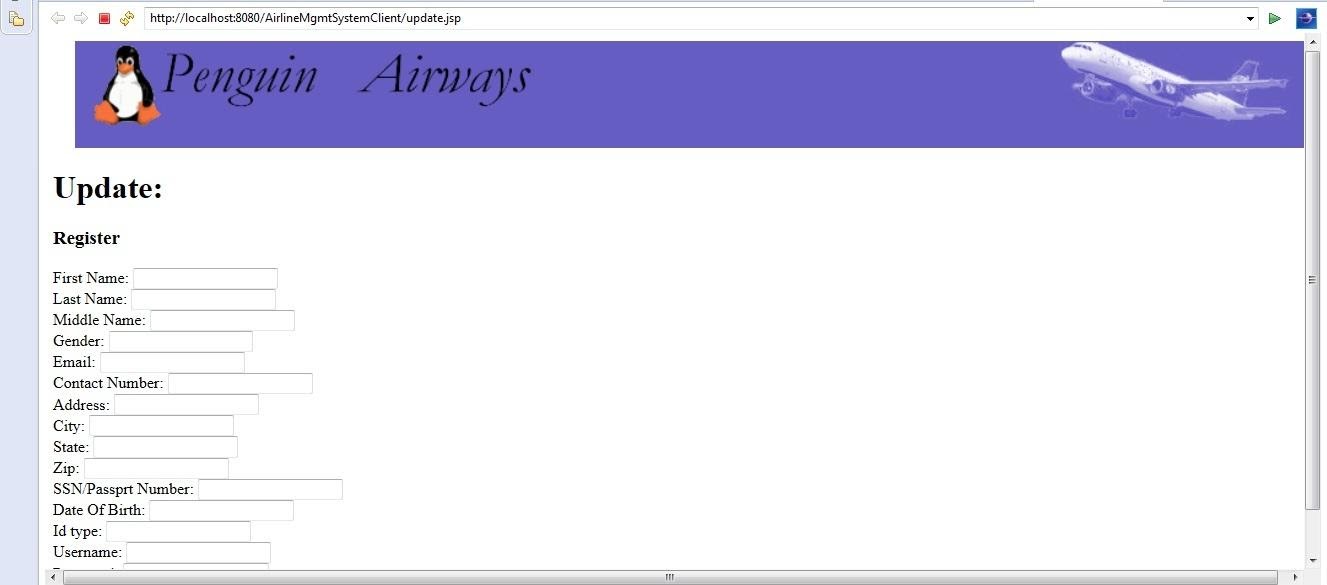
Login Page



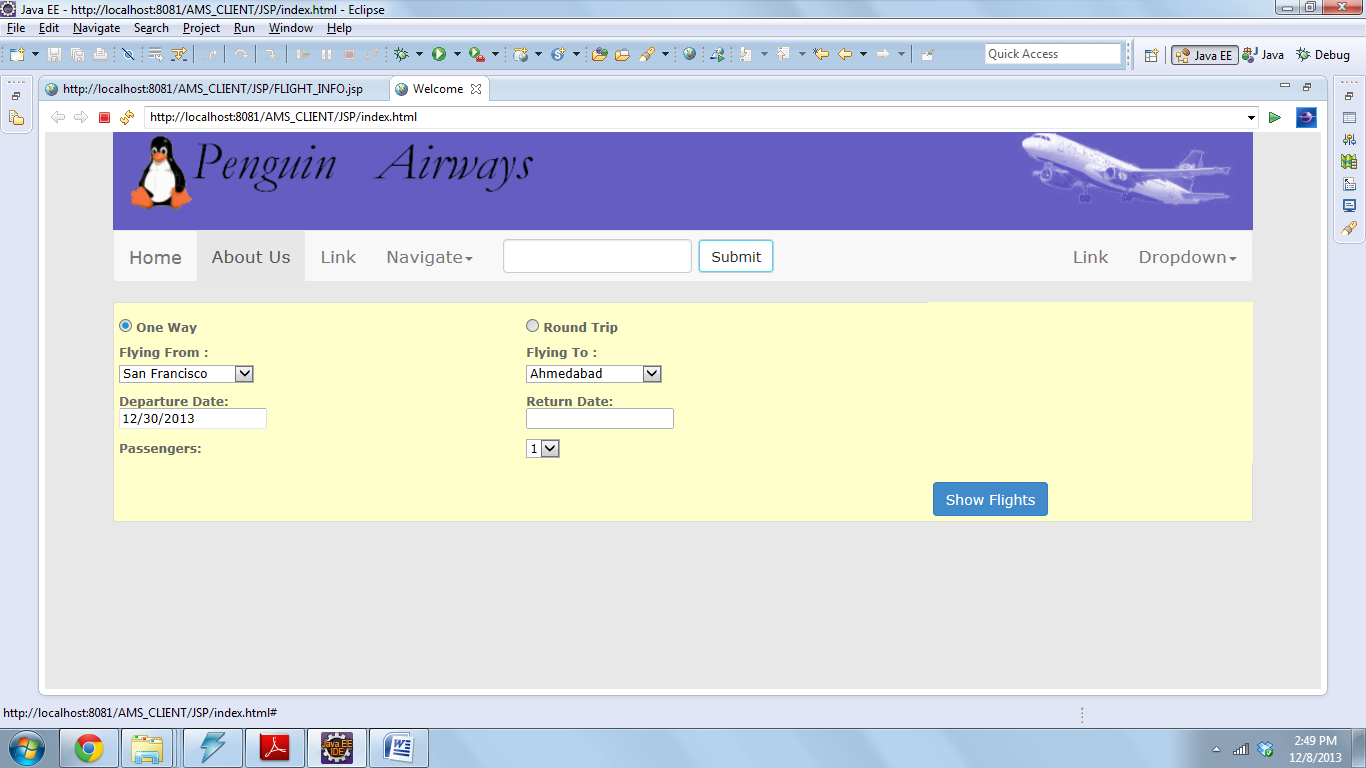
Sign Up

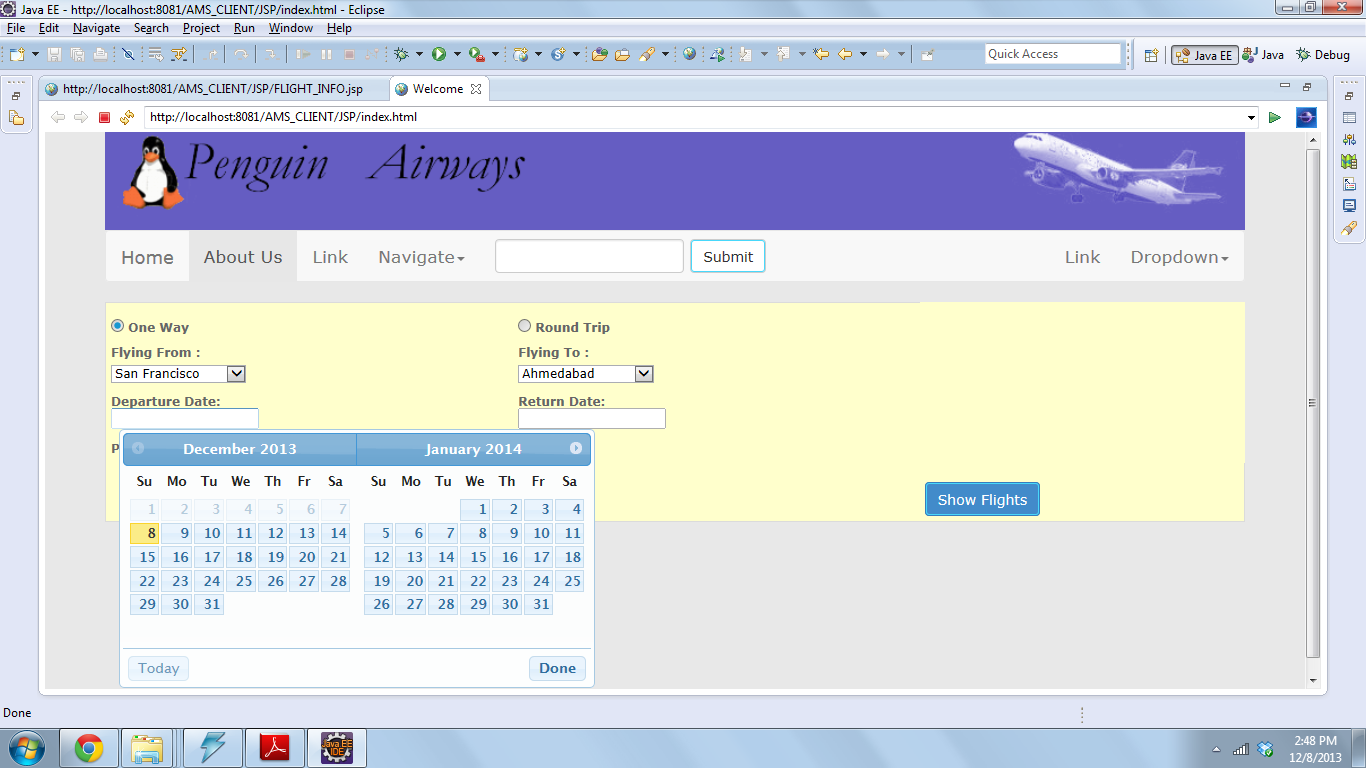


Update Customer

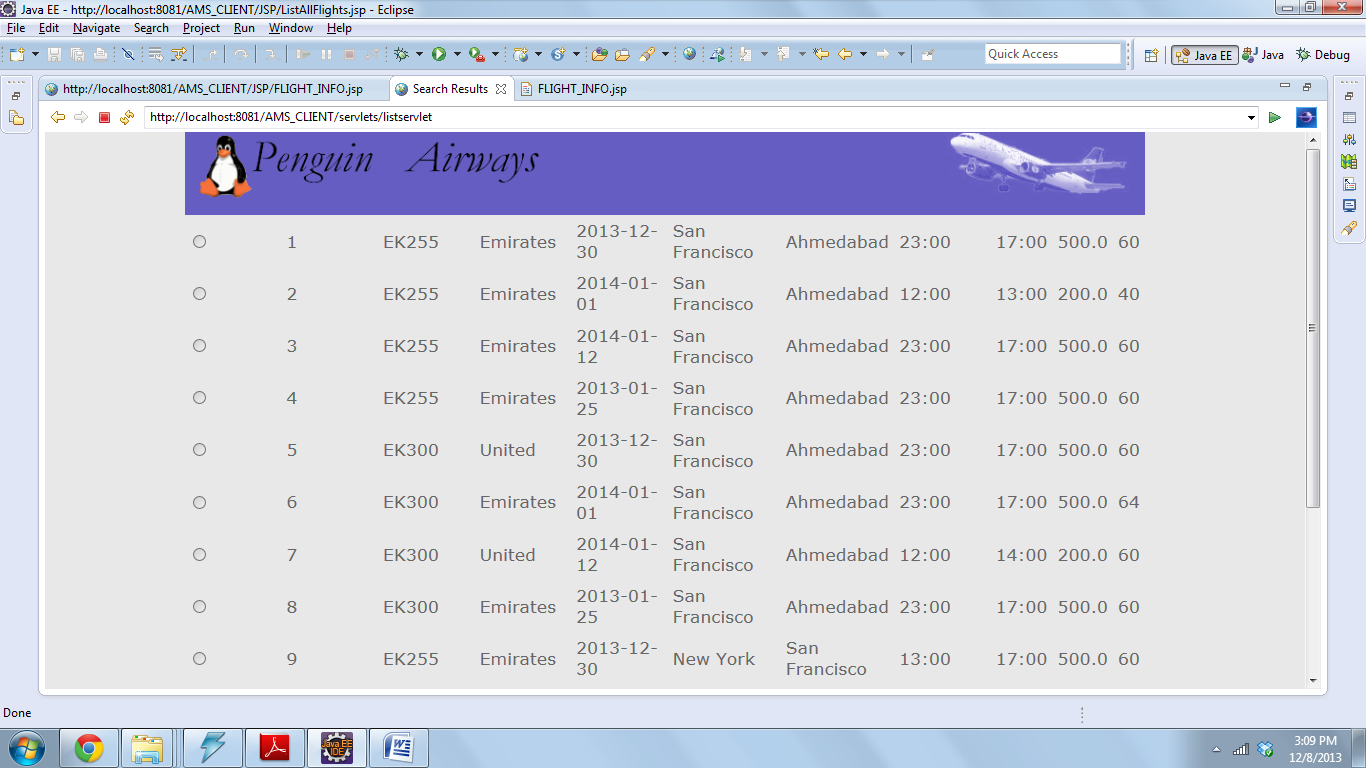


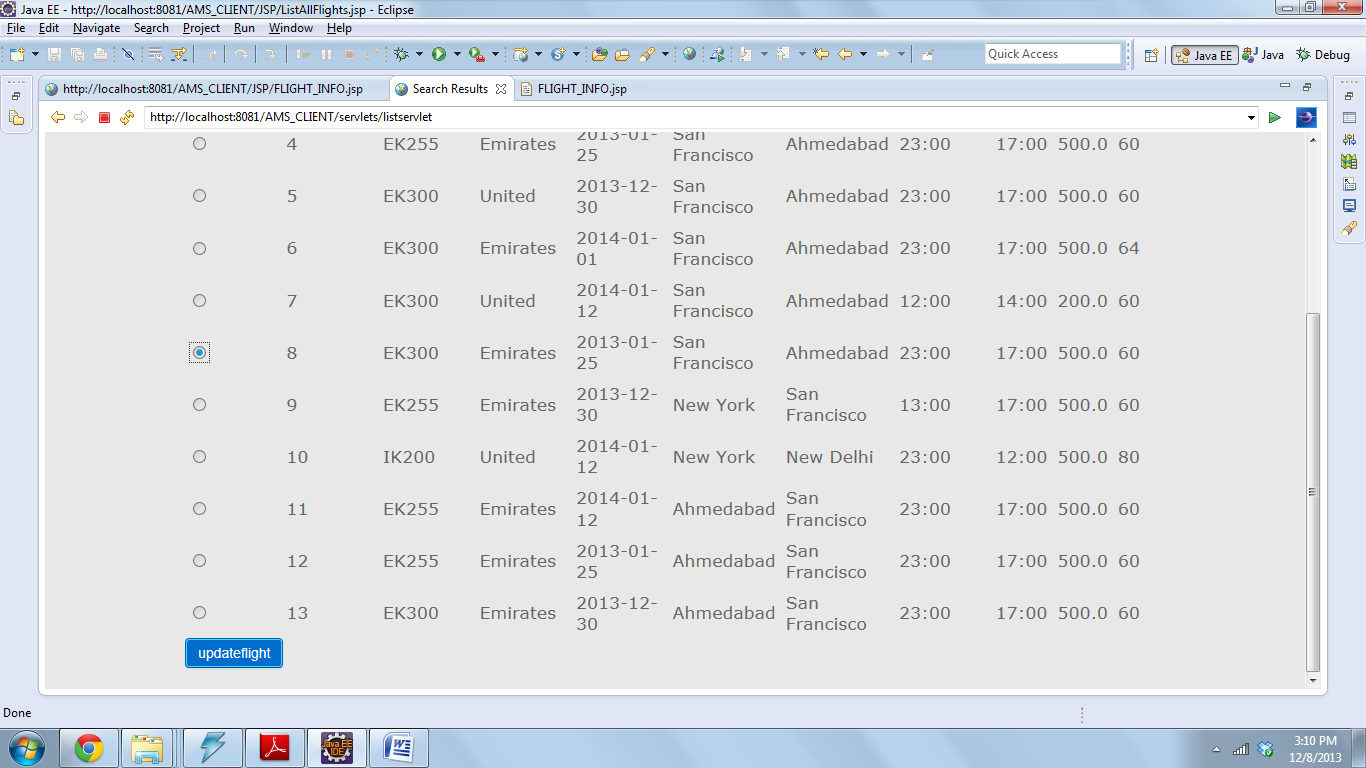
Search Flight



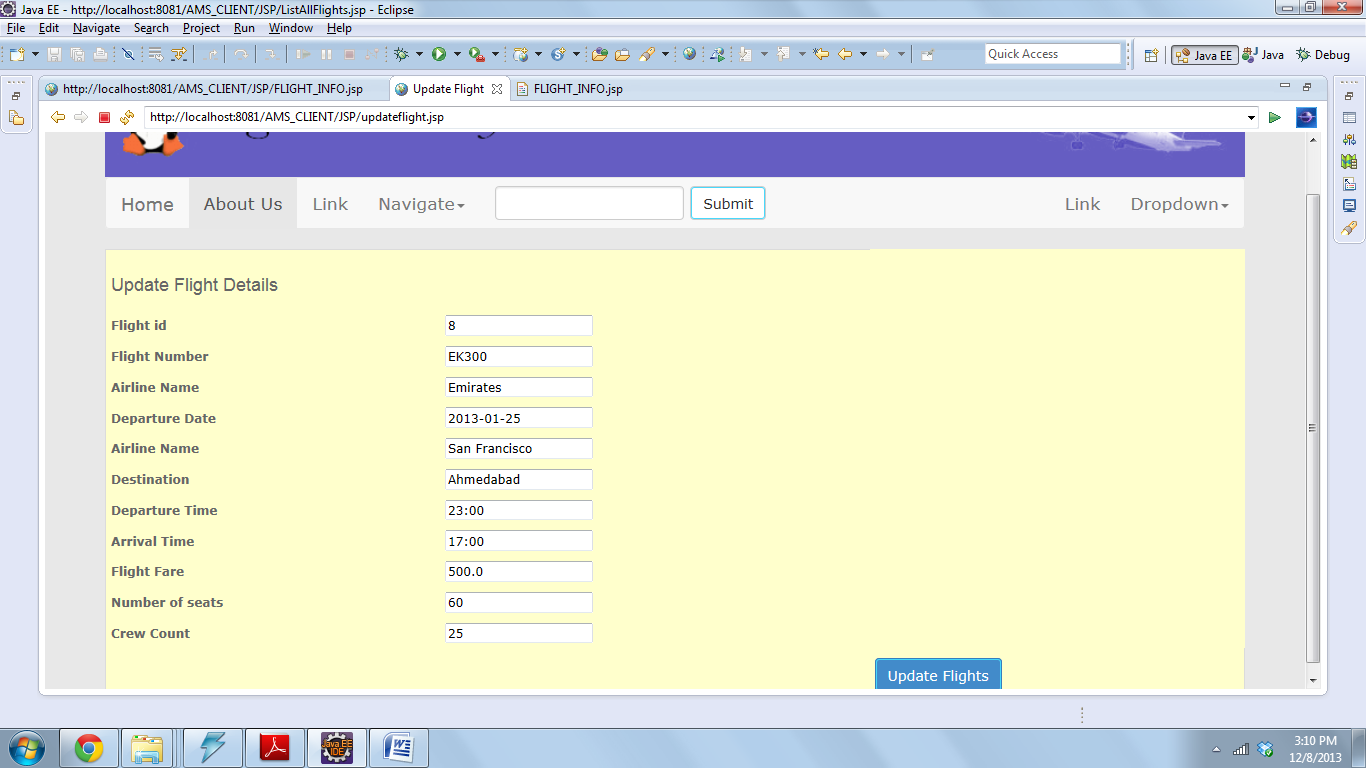


List Flight

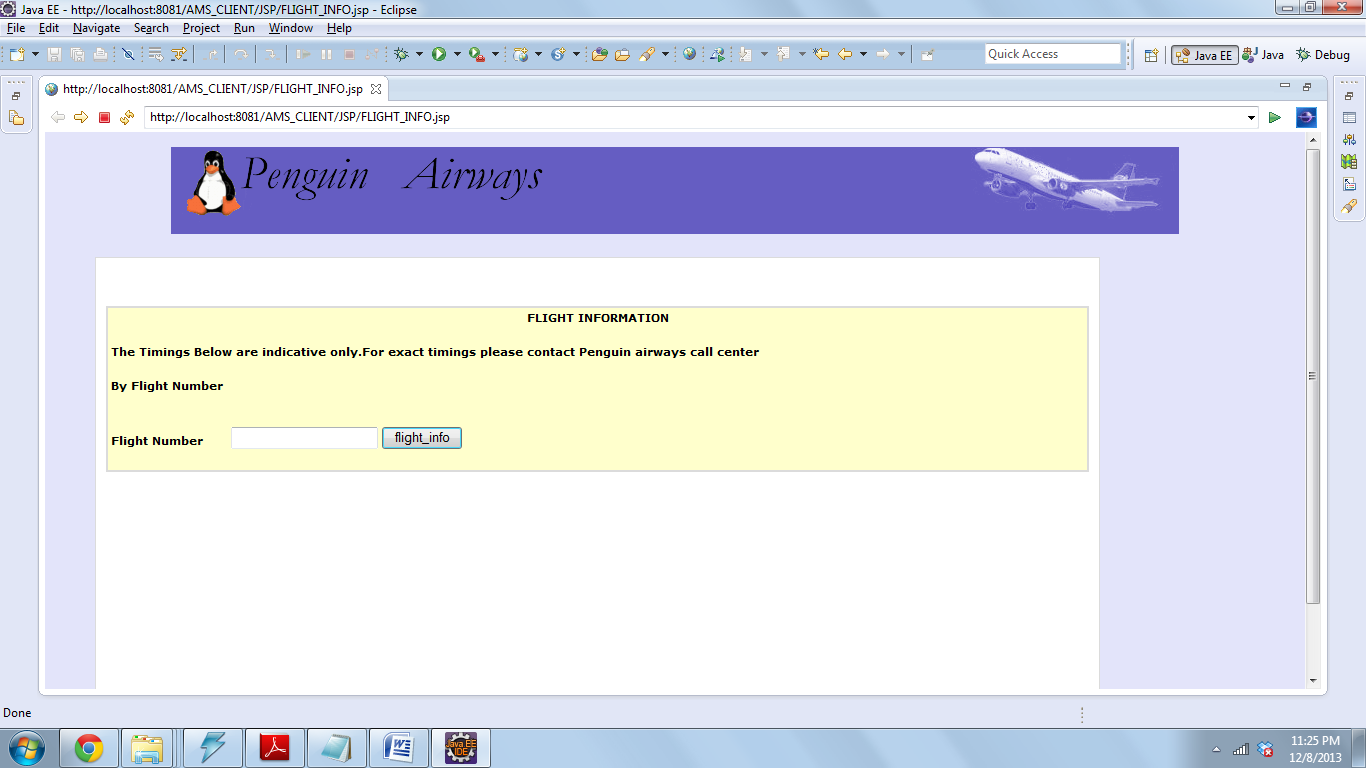


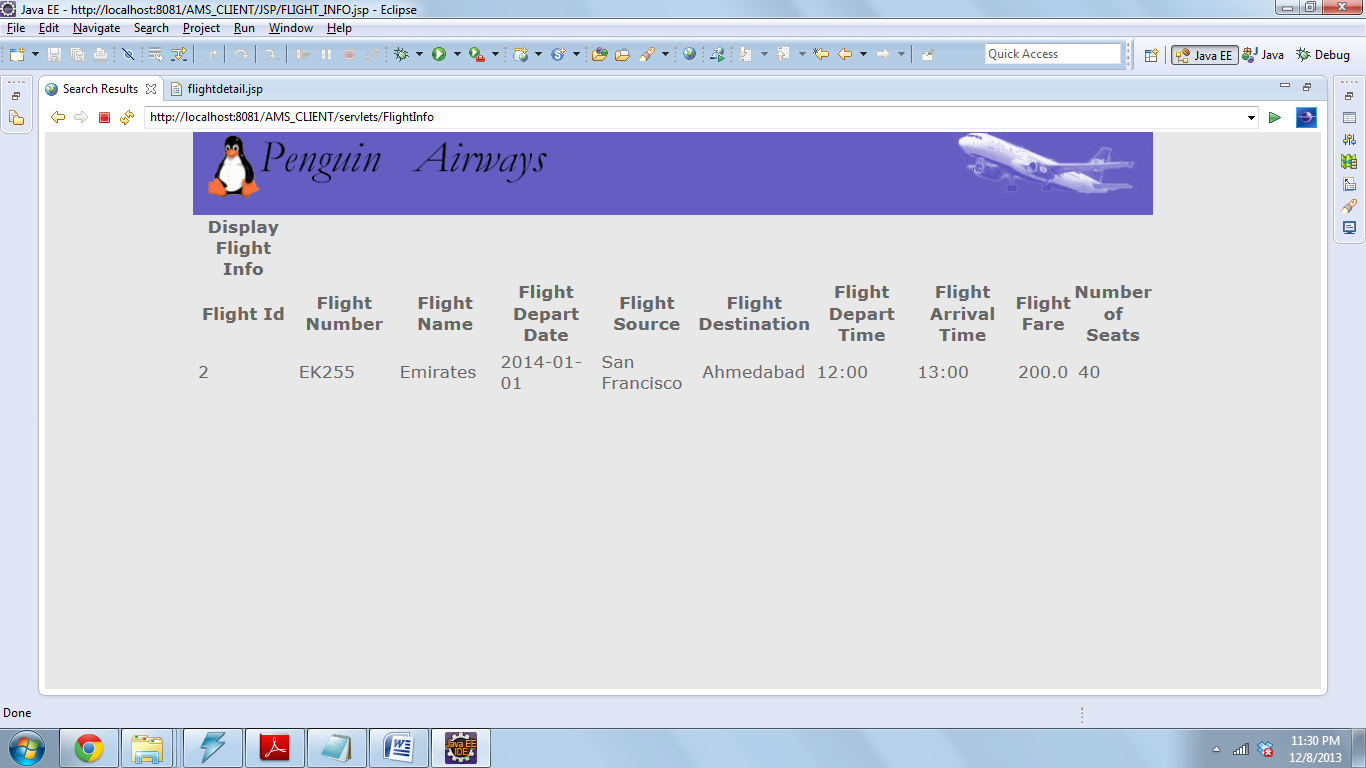


Update Flight

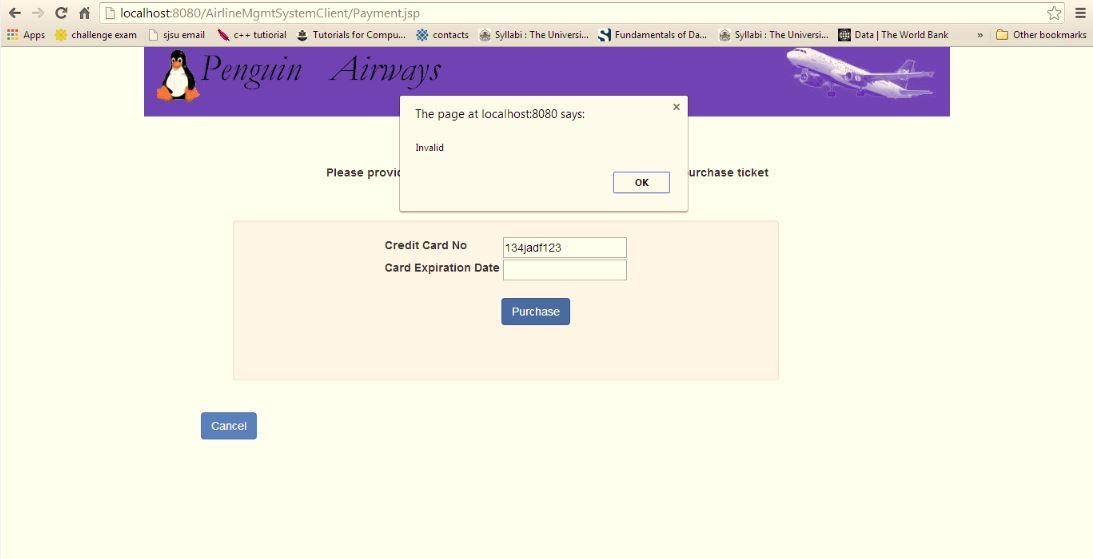


Display Flight Info

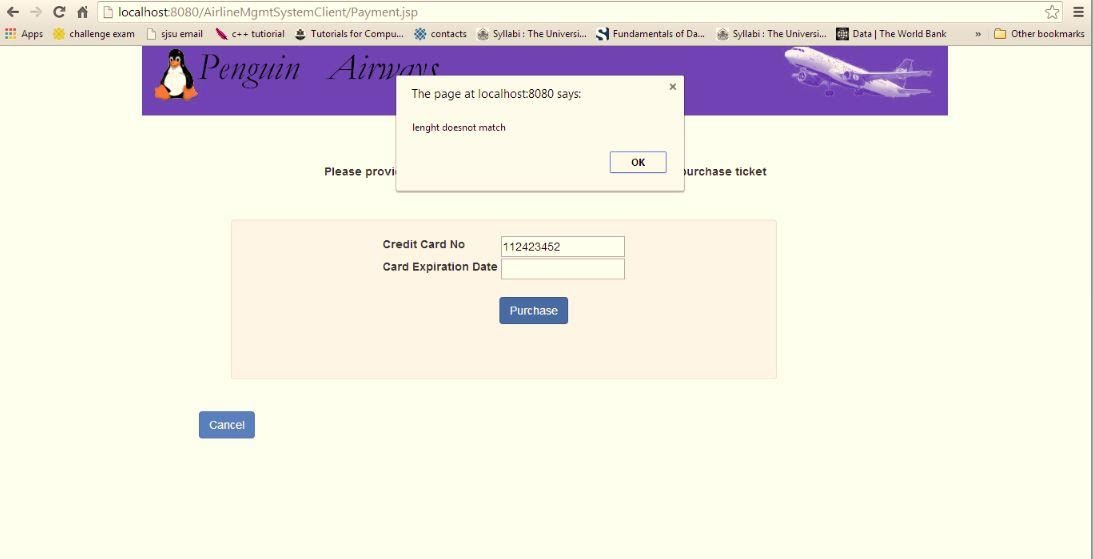




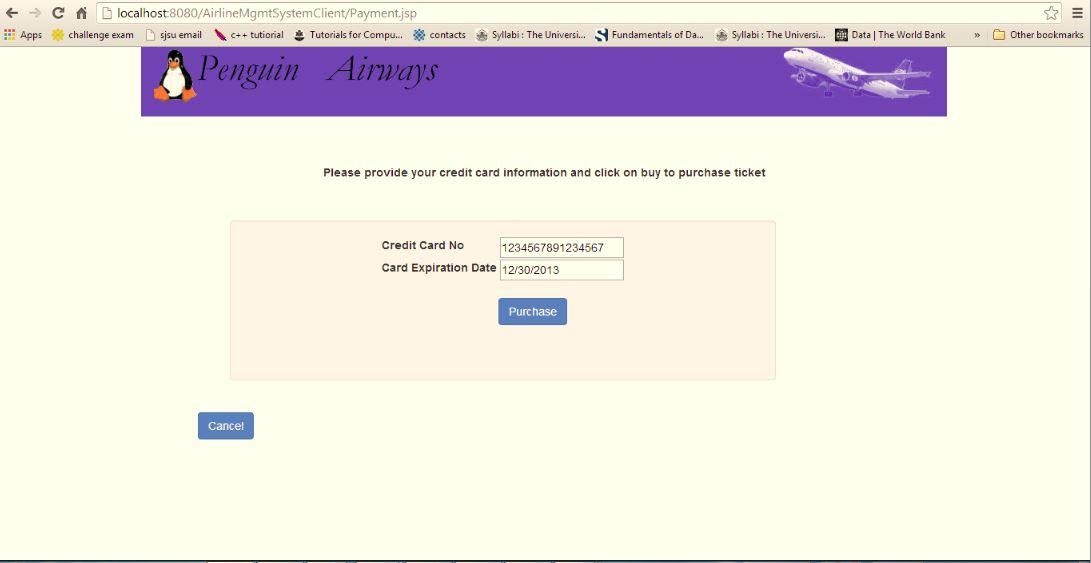
Credit Card 1



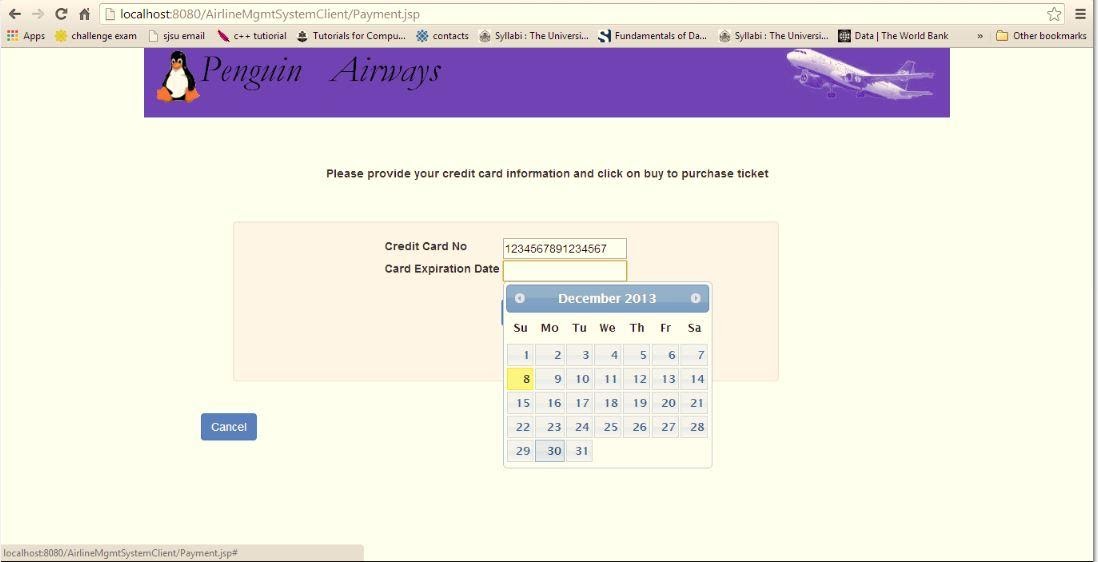
Credit Card 2

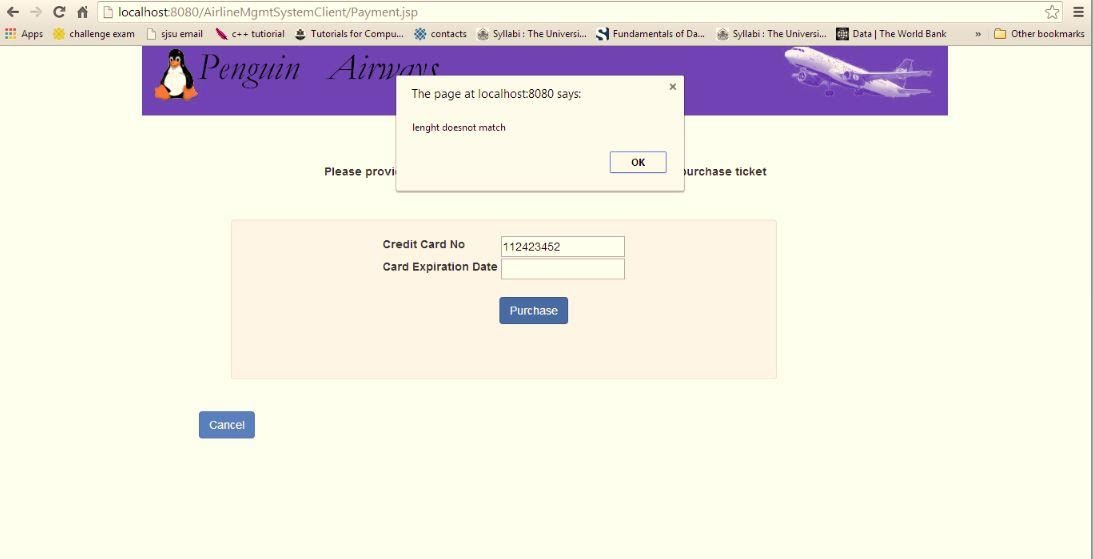


Purchase 1



Purchase 2



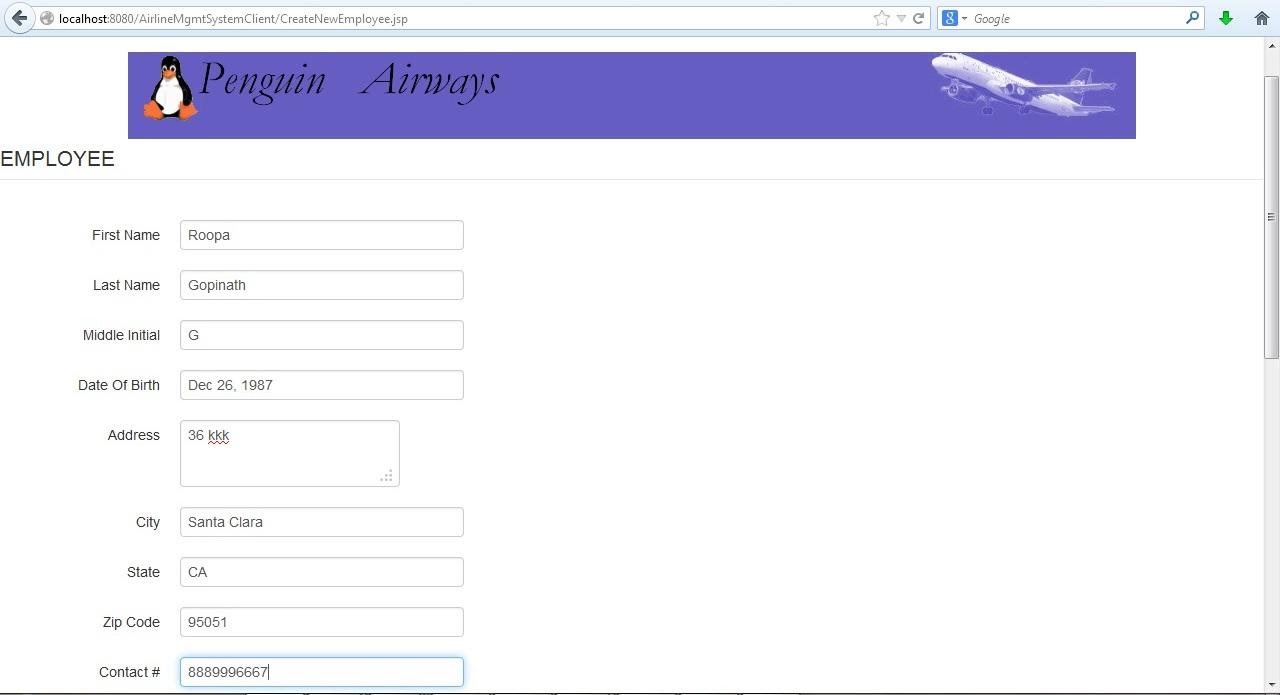


e Ticket

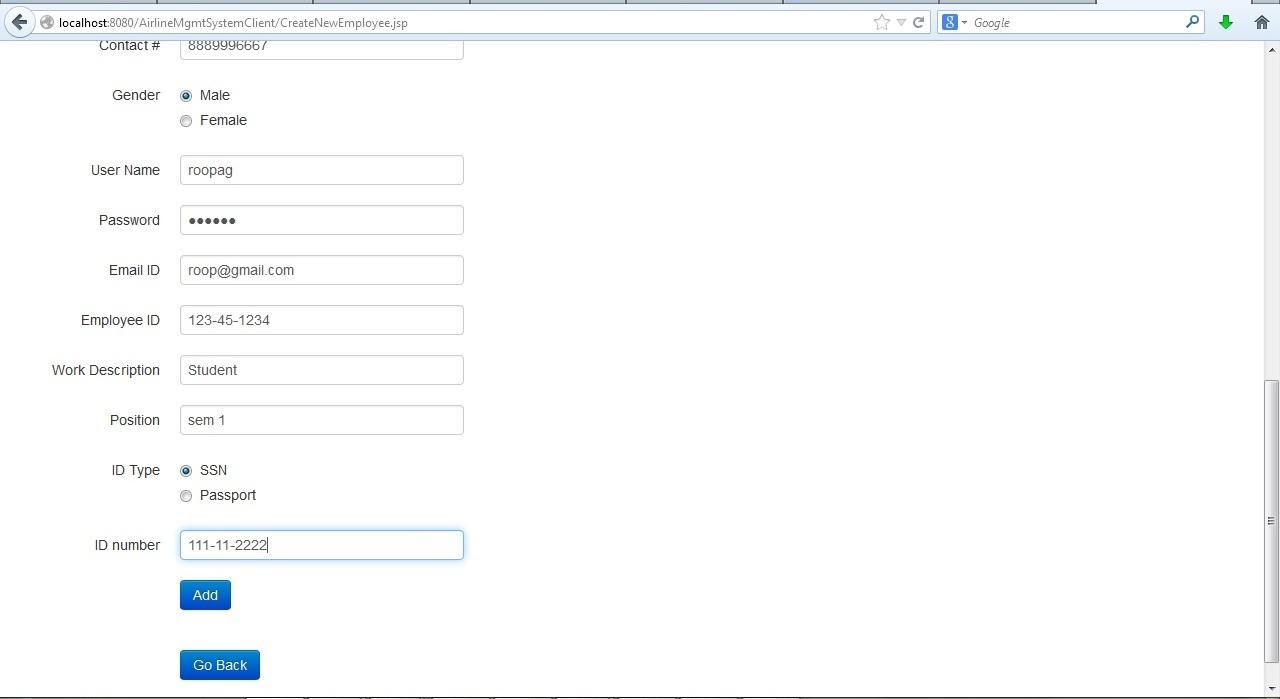
Ticket

Admin Welcome Page – Employee.jsp

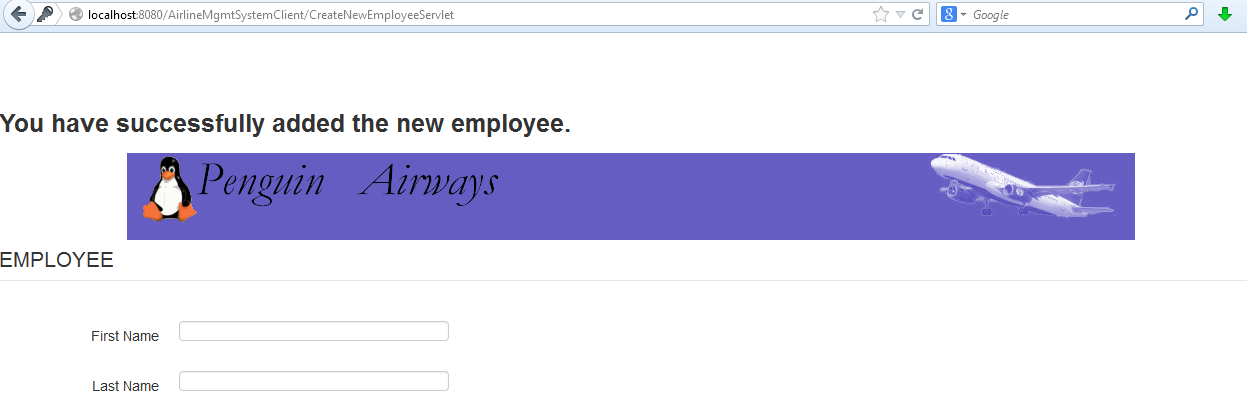
Create Employee 1



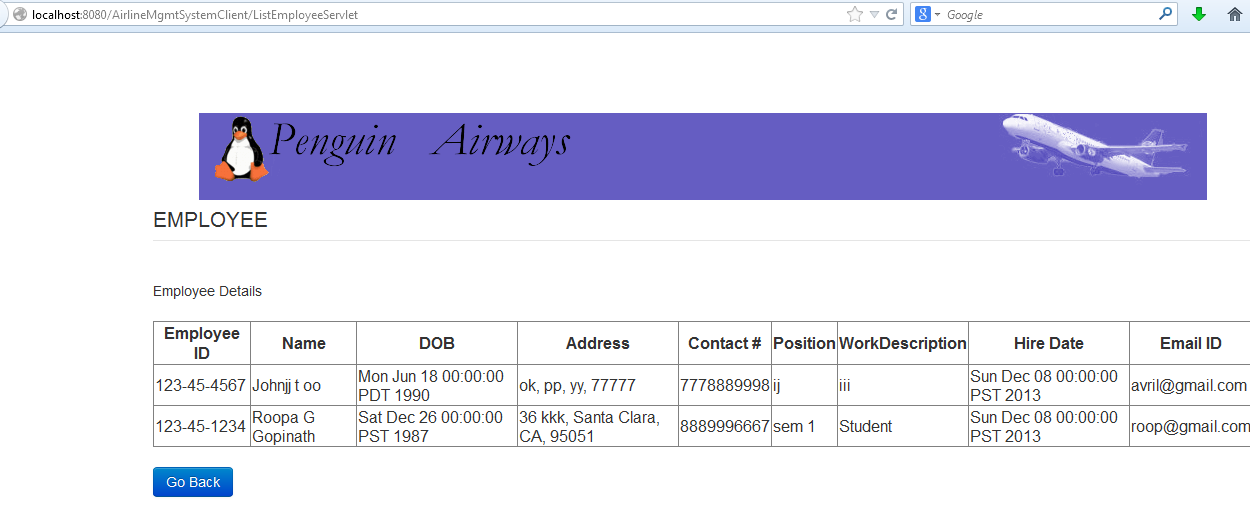
Create Employee 2



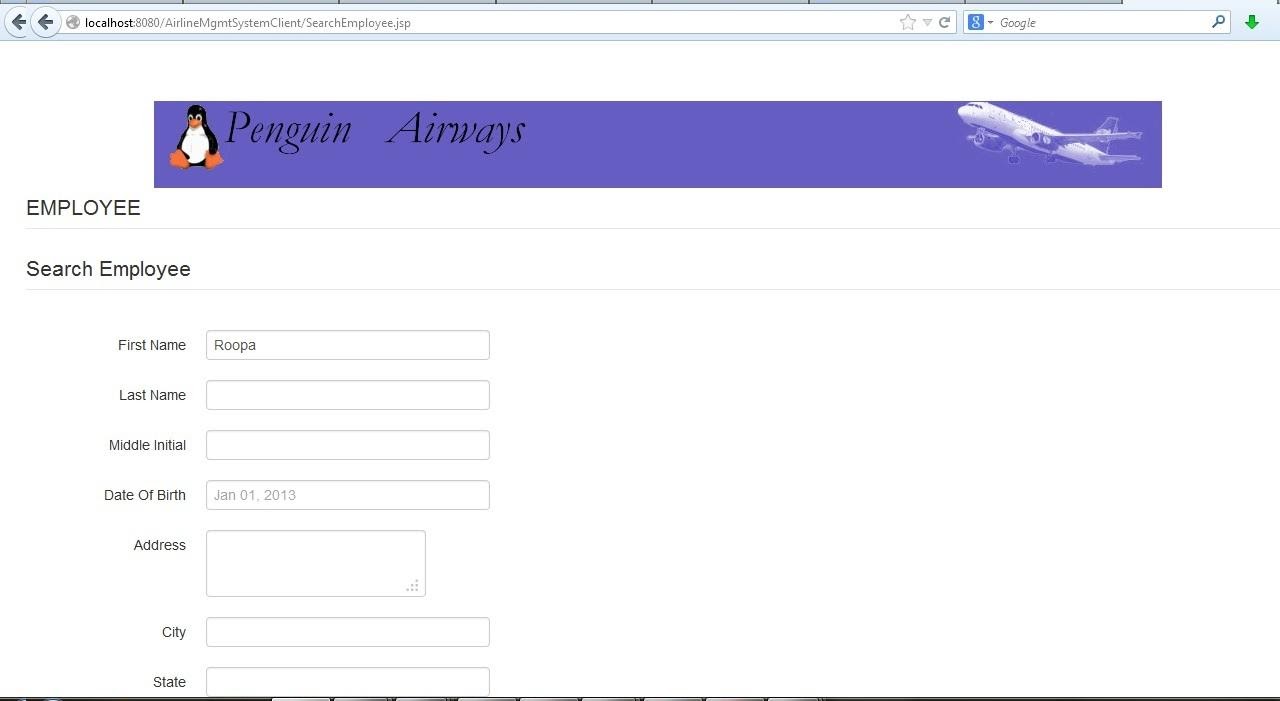
Create Employee 3



List Employee

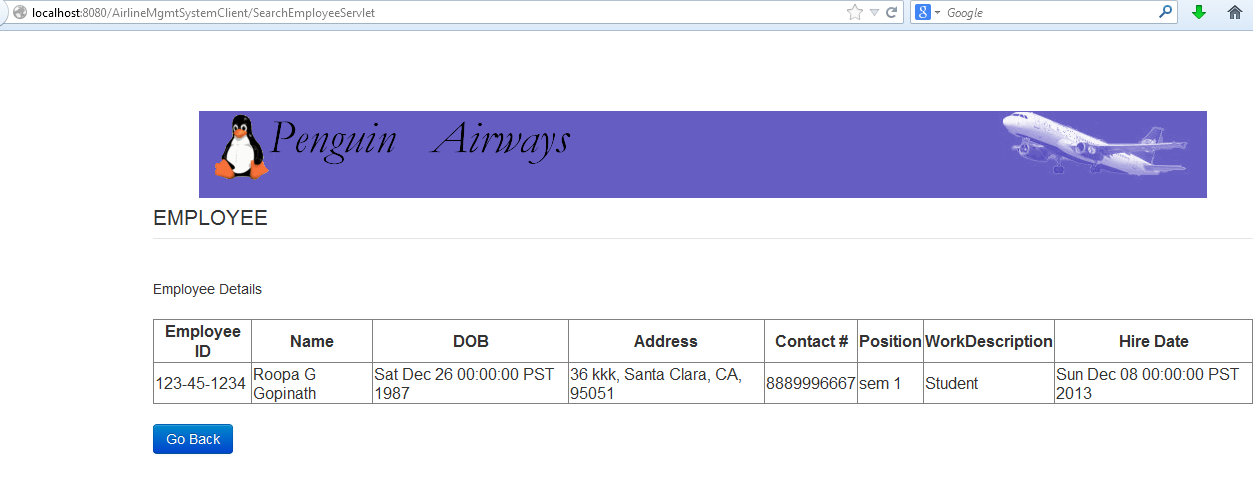


Search Employee

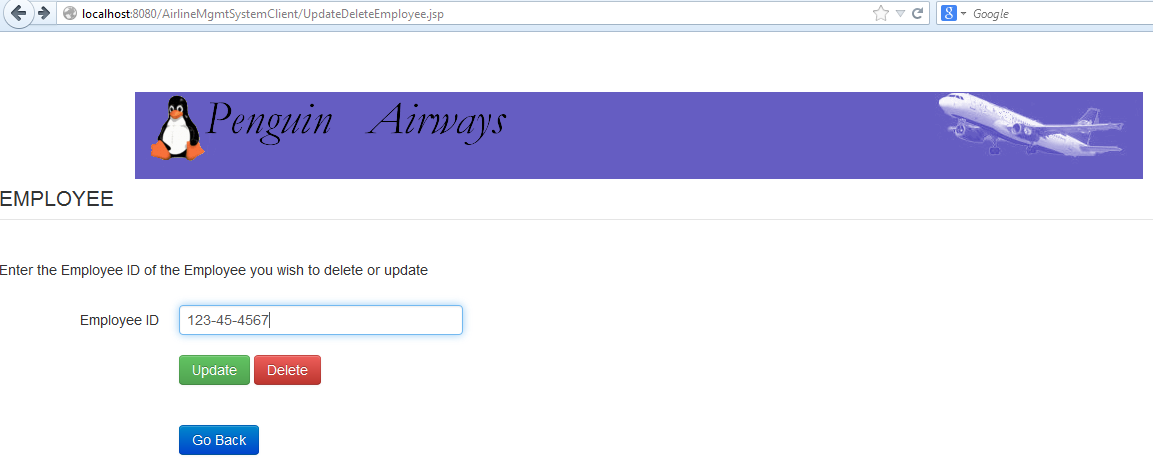
****

Search Employee 2

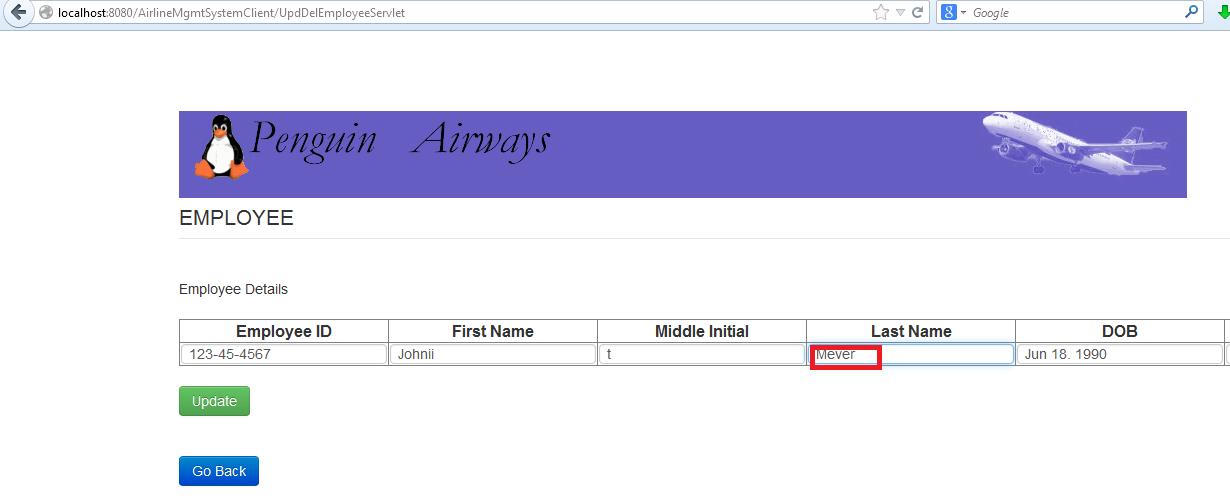
Search Result

****

Update/Delete option

****

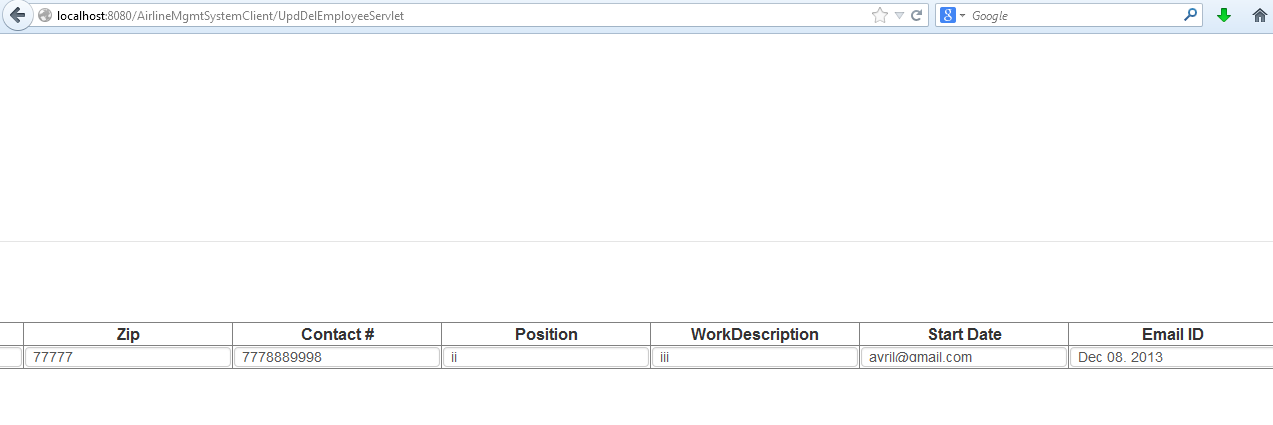
Update 1

****

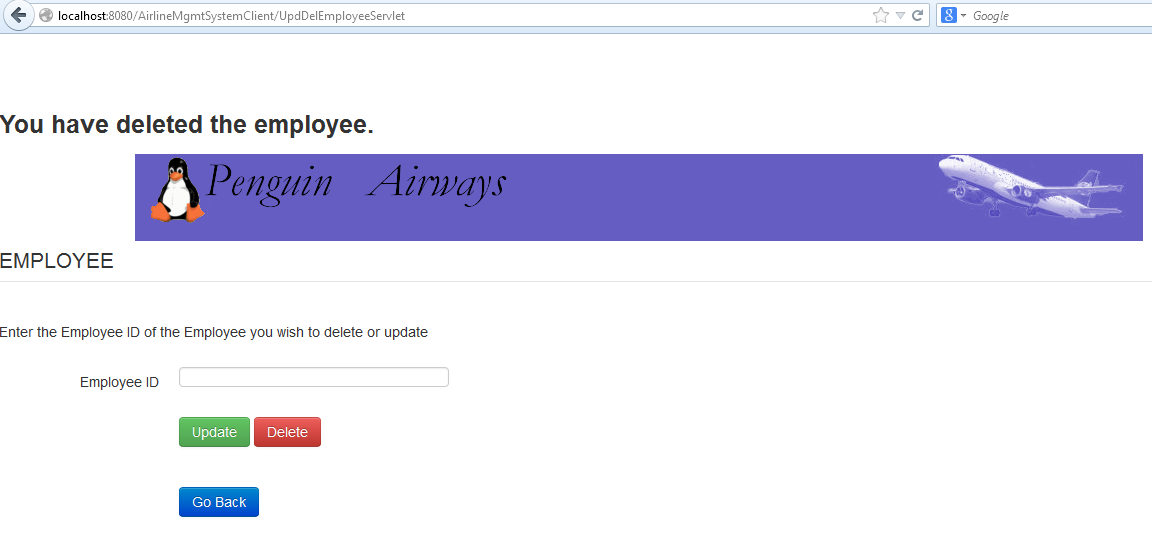
Update 2



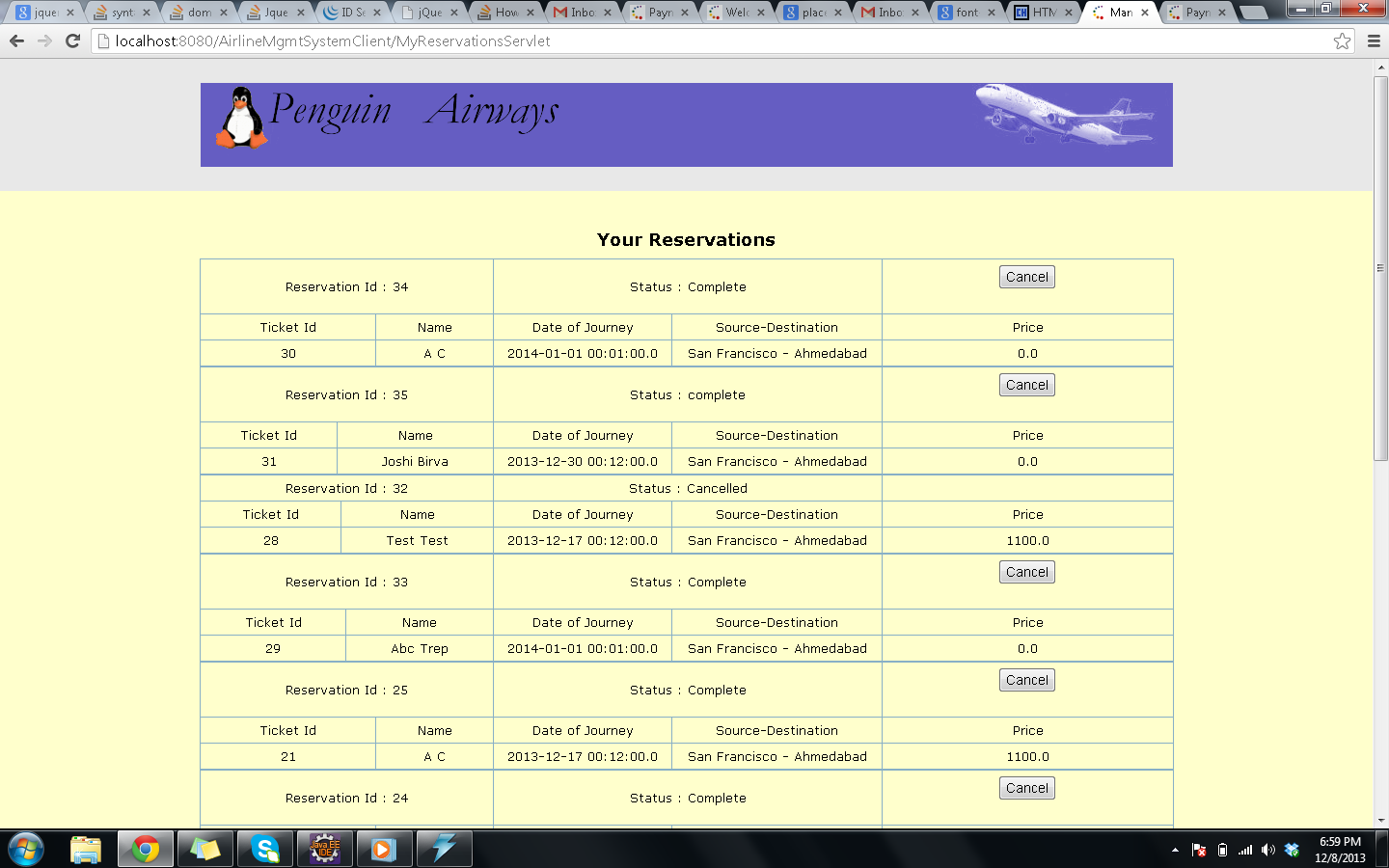
Update 3



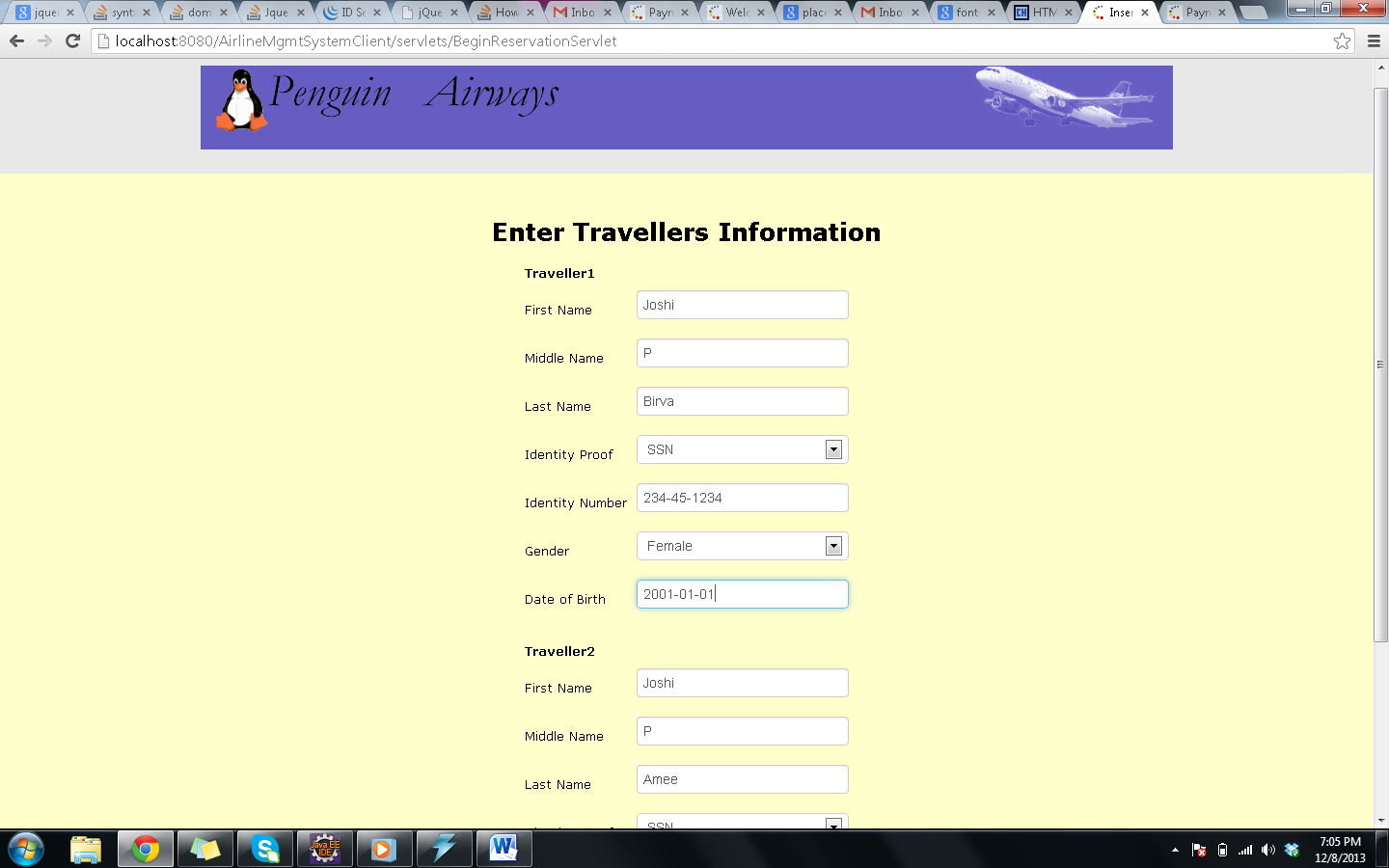
Delete Employee



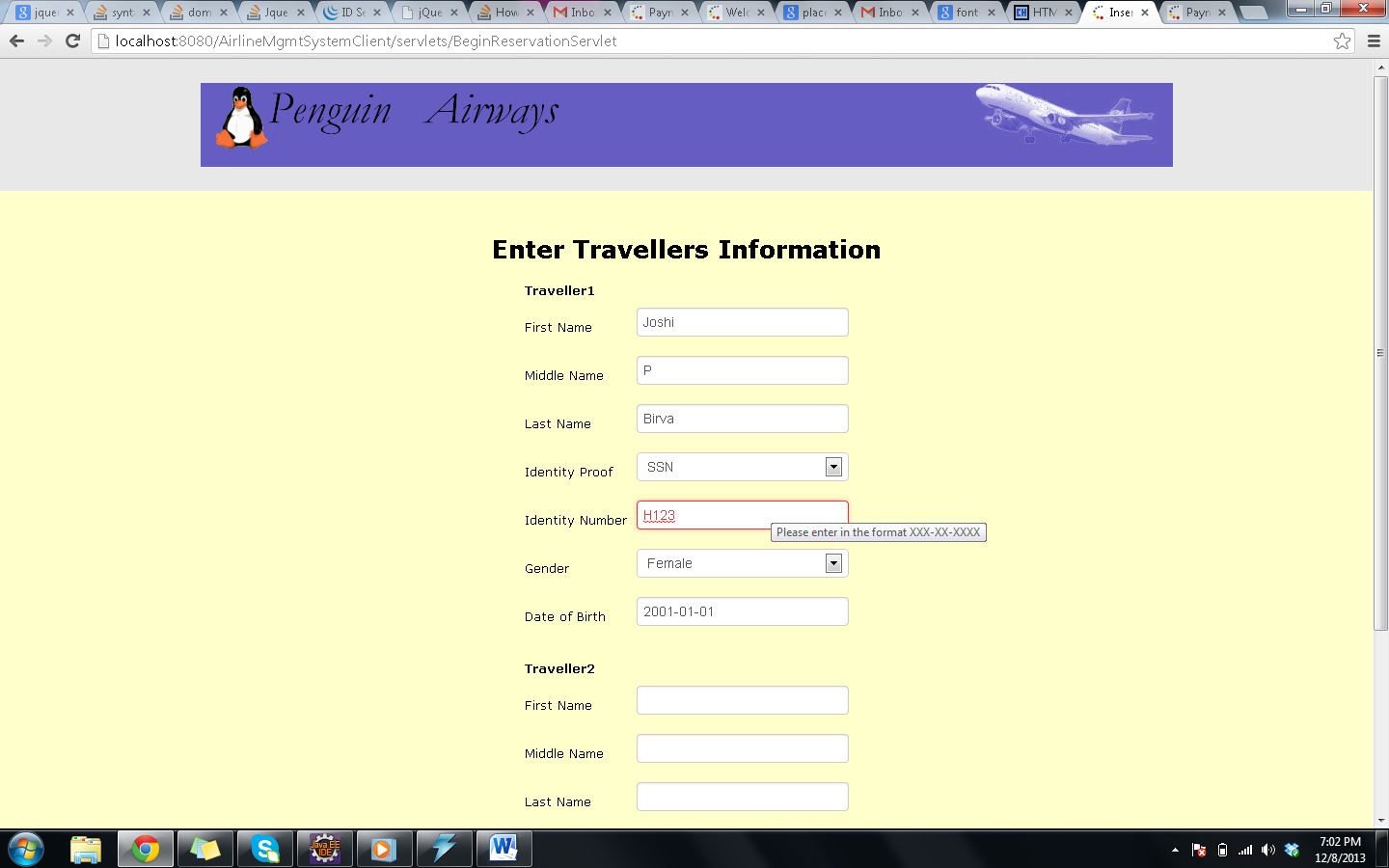
Manage Reservations



Traveller

****

Traveller Validation



**Database creation Scripts**

CREATE TABLE IF NOT EXISTS Person(

person\_id int(11) NOT NULL AUTO\_INCREMENT,

id\_no varchar(32) NOT NULL,

id\_type varchar(32) NOT NULL,

first\_name varchar(45) NOT NULL,

last\_name varchar(45) NOT NULL,

middle\_initial char(1) NOT NULL,

gender varchar(32) DEFAULT NULL,

date\_of\_birth timestamp NOT NULL DEFAULT '2013-11-20 00:00:00',

contact\_no varchar(45) NOT NULL,

address varchar(100) DEFAULT NULL,

city varchar(45) DEFAULT NULL,

state varchar(45) DEFAULT NULL,

zipcode varchar(10) DEFAULT NULL,

creation\_date timestamp NOT NULL DEFAULT '2013-11-20 00:00:00',

created\_by varchar(45) NOT NULL,

last\_updated timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,

last\_updated\_by varchar(45) NOT NULL,

username varchar(45) NOT NULL,

passWrd varchar(45) NOT NULL,

email varchar(45) NOT NULL,

PRIMARY KEY(person\_id)

);

CREATE TABLE IF NOT EXISTS `traveller` (

  `traveller\_id` int(11) NOT NULL AUTO\_INCREMENT,

  `id\_no` varchar(80) NOT NULL,  /\* Restrict format xxx-xx-xxxx for ssn from code \*/

  `id\_type` varchar(60) NOT NULL, /\* SSN or passport \*/

`person\_id` int(10) DEFAULT NULL,

  `creation\_date` timestamp NOT NULL DEFAULT '2013-11-20 00:00:00',

  `created\_by` varchar(45) NOT NULL,

  `last\_updated` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,

  `last\_updated\_by` varchar(45) NOT NULL,

PRIMARY KEY (`traveller\_id`),

  UNIQUE KEY `identity` (`id\_no`, `id\_type`),

  FOREIGN KEY (`person\_id`) references person(person\_id)

  ON DELETE CASCADE

) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=latin1;

CREATE TABLE IF NOT EXISTS Employee (

  Employee\_id varchar(15) NOT NULL,

  id\_no varchar(32) NOT NULL,

  id\_type varchar(32) NOT NULL,

  person\_id int(10) DEFAULT NULL,

  creation\_date timestamp NOT NULL DEFAULT '2013-11-20 00:00:00',

  created\_by varchar(45) NOT NULL,

  last\_updated timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,

  last\_updated\_by varchar(45) NOT NULL,

  work\_description varchar(45),

  position varchar(45),

  hire\_date date,

  end\_date date,

  PRIMARY KEY (employee\_id),

  UNIQUE KEY identity (id\_no, id\_type),

  FOREIGN KEY (person\_id) references person(person\_id)

  ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS `reservations` (

  `reservation\_id` int(10) unsigned NOT NULL AUTO\_INCREMENT,

`customer\_id` int(10) NOT NULL,

  `no\_of\_people` int(10) unsigned NOT NULL,

  `type\_of\_journey` varchar(45) NOT NULL,

  `status` varchar(32) NOT NULL DEFAULT 'InCheckout',

  `creation\_date` timestamp NOT NULL DEFAULT '2013-11-20 00:00:00',

  `created\_by` varchar(45) NOT NULL,

  `last\_updated` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

  `last\_updated\_by` varchar(45) NOT NULL,

  PRIMARY KEY (`reservation\_id`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE IF NOT EXISTS `reservation\_details` (

  `reservation\_dtl\_id` int(10) unsigned NOT NULL AUTO\_INCREMENT,

  `reservation\_id` int(10) unsigned NOT NULL,

  `traveller\_id` int(11) NOT NULL,

  `flight\_id` varchar(45) NOT NULL,

  `seat\_no` varchar(45) NOT NULL,

  `date\_of\_journey` timestamp NOT NULL DEFAULT '2013-12-01 00:00:00',

  `travel\_class` varchar(45) NOT NULL,

  `price` float(10,2) DEFAULT 0.0,

  `tax` float(10,2) DEFAULT 0.0,

  `status` varchar(45) NOT NULL,

  `boarding\_status` varchar(45) NOT NULL,

  `creation\_date` timestamp NOT NULL DEFAULT '2013-11-20 00:00:00',

  `created\_by` varchar(45) NOT NULL,

  `last\_updated` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,

  `last\_updated\_by` varchar(45) NOT NULL,

  PRIMARY KEY (`reservation\_dtl\_id`),

    FOREIGN KEY (`traveller\_id`) references traveller(traveller\_id),

  FOREIGN KEY (`reservation\_id`) references reservations(reservation\_id)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

create table if not exists FlightDetails

(

Flight\_Id int(11) AUTO\_INCREMENT,

Flight\_Number varchar(5),

Airline\_Name varchar(50),

Depart\_date DATE,

Flight\_Source varchar(50),

Flight\_Destination varchar(50),

Flight\_Depart\_Time varchar(8),

Flight\_Arr\_Time varchar(8),

Flight\_fare varchar(10),

No\_Of\_Seats integer,

Crew\_count int(11),

PRIMARY KEY(Flight\_Id));

CREATE TABLE IF NOT EXISTS `payment` (

  `payId` int(11) NOT NULL,

  `ReservationId` int(11) DEFAULT NULL,

  `FlightId` int(11) DEFAULT NULL,

  `FlightNo` int(11) DEFAULT NULL,

  `PersonId` int(11) DEFAULT NULL,

  `PaymentDate` date DEFAULT NULL,

  `Amount` int(11) DEFAULT NULL,

  `PaymentType` varchar(45) DEFAULT NULL,

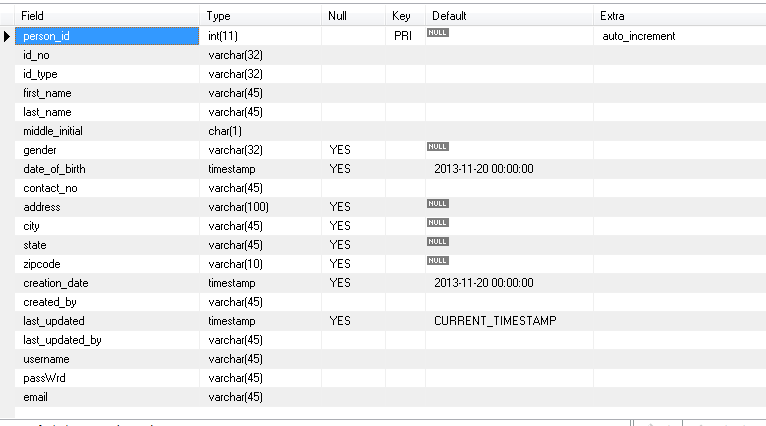
  PRIMARY KEY (`payId`)

)

**Screen Capture showing database tables**

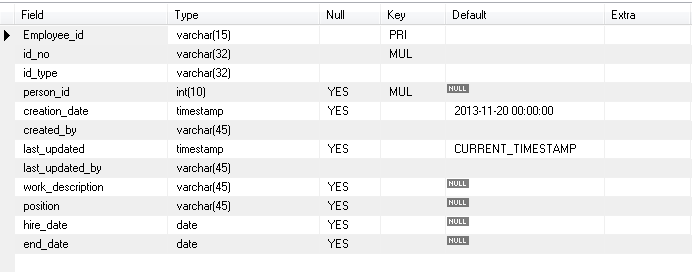
**PERSON table**

**mysql> desc Person**

****

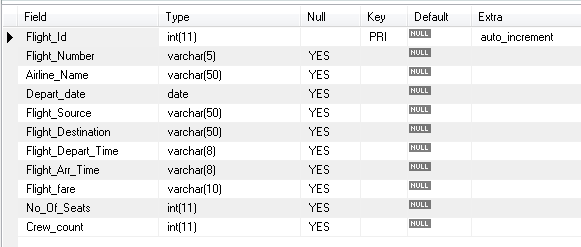
**EMPLOYEE Table**

**mysql> desc Employee**

****

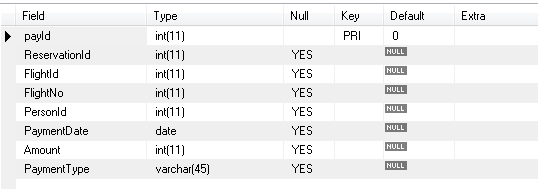
**FLIGHTDETAILS table**

**mysql> desc flightDetails;**

****

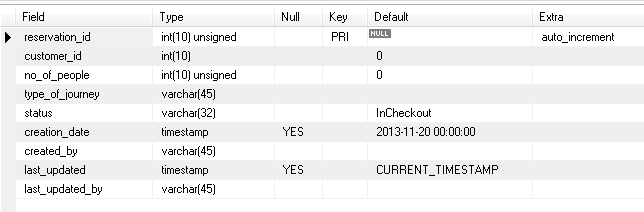
**PAYMENT table**

**mysql> desc Payment;**

****

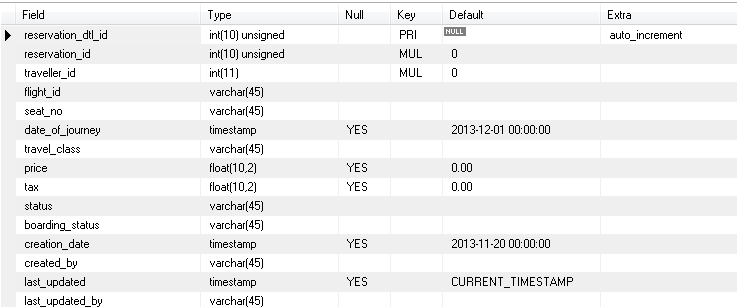
**RESERVATIONS table**

**mysql> desc Reservations;**

****

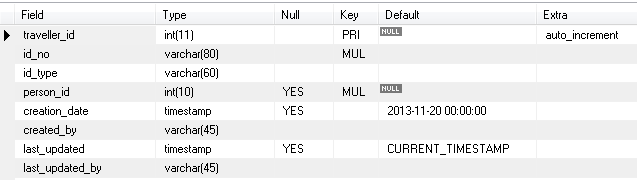
**RESERVATION\_DETAILS table**

**mysql> desc Reservation\_Details;**

****

**TRAVELLER table**

**mysql> desc Traveller;**

****

**Other Requirements handled**

1. **Maintain a pool of DB connections**

Connection pooling is done where all the modules access the database through already existing connection objects (We have 20 such objects).

Note: Please refer to “How heavyweight resources are handled” section in Page 8 of this report

1. **Cache entity lookups**

We have used lazy loading for caching. On the first request for a flight search, the results are cached in a map with source and destination as key. The search results are mapped to bean files and are stored against the key. When a flight is updated, the map is looked upon and the object in the map is updated accordingly.

Note: Please refer to “Object management policy” section in Page 5

1. **To increase reliability**

**-**We have server side validations like null checks

**-**Proper error handling and displaying appropriate error messages

**-**The database has tables with referential integrity maintained with the help of an int field i.e. person\_id. This prevents incorrect entry into tables.

1. **Scalability**

Current implementation has a maximum of 20 database connection objects. If we need more connections in case more functionalities have to be implemented which need database access, it can be easily scaled by just changing the number of active connections in the code.

The Person and Employee tables have 5000 records which re handled by the code.

1. **Exceptions /Failure Modes**

Exceptions are handled on the server side by using try and catch blocks for all methods and appropriate error messages are displayed.

1. **Validations**

* Basic client side validations include incorrect datatype check, mandatory/non-mandatory fields handling, pattern matching for specific fields like zip code which is of pattern [0-9][0-9][0-9][0-9][0-9], Employee ID which is of the pattern of US Social security number etc
* Correct navigation from one page to the other is verified.
* Module wise testing and end to end integration testing is done

**7. JMS**

JMS is used for publishing flight status to all the clients. The code listing of the client and Server is listed below,

//Client code

import java.util.Properties;

import java.util.Scanner;

import java.util.regex.Pattern;

import javax.jms.Connection;

import javax.jms.ConnectionFactory;

import javax.jms.JMSException;

import javax.jms.Message;

import javax.jms.MessageConsumer;

import javax.jms.MessageListener;

import javax.jms.MessageProducer;

import javax.jms.Queue;

import javax.jms.Session;

import javax.jms.TextMessage;

import javax.jms.Topic;

import javax.naming.Context;

import javax.naming.InitialContext;

import javax.naming.NamingException;

public class Client {

private Connection connection;

private Session session;

private MessageConsumer consumer, queueConsumer;

private Queue StatusQueue, replyQueue;

static String reply =null;

public Client()

{

try

{

Properties properties = new Properties();

properties.put(Context.INITIAL\_CONTEXT\_FACTORY, "org.jnp.interfaces.NamingContextFactory");

properties.put(Context.URL\_PKG\_PREFIXES, "org.jnp.interfaces");

properties.put(Context.PROVIDER\_URL, "localhost");

InitialContext jndi = new InitialContext(properties);

ConnectionFactory conFactory = (ConnectionFactory)jndi.lookup("XAConnectionFactory");

connection = conFactory.createConnection();

session = connection.createSession( false, Session.AUTO\_ACKNOWLEDGE );

StatusQueue = (Queue)jndi.lookup("queue/StatusQueue");

connection.start();

}

catch(NamingException NE)

{

System.out.println("Naming Exception: "+NE);

}

catch(JMSException JMSE)

{

System.out.println("JMS Exception: "+JMSE);

}

}

public void FlightInfo(int flightId) throws JMSException

{

System.out.println("inside clinet startrs");

MessageProducer MP = session.createProducer(StatusQueue);

TextMessage TM = session.createTextMessage("Status,"+flightId);

replyQueue=session.createTemporaryQueue();

consumer=session.createConsumer(replyQueue);

TM.setJMSReplyTo(replyQueue);

MP.send(TM);

TextMessage Reply = (TextMessage)consumer.receive();

reply=Reply.getText();

System.out.println("and reply is "+reply);

System.out.println("inside login clinet ends");

}

public static void main(String[] args) {

Client cl=new Client();

try {

cl.FlightInfo(1);

} catch (JMSException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

//Server code

import java.util.Properties;

import javax.jms.Connection;

import javax.jms.ConnectionFactory;

import javax.jms.Destination;

import javax.jms.JMSException;

import javax.jms.Message;

import javax.jms.MessageConsumer;

import javax.jms.MessageListener;

import javax.jms.MessageProducer;

import javax.jms.Queue;

import javax.jms.Session;

import javax.jms.TextMessage;

import javax.naming.Context;

import javax.naming.InitialContext;

import javax.naming.NamingException;

public class Server implements MessageListener

{

private Connection connection;

private Session session;

private MessageConsumer consumer;

private Queue StatusQueue;

private FlightStatus status=new FlightStatus();

public void sendReply(Message request, String returnData)

{

System.out.println("Inside send reply");

try

{

MessageProducer MP = session.createProducer(null);

Destination reply = request.getJMSReplyTo();

TextMessage TM = session.createTextMessage();

TM.setText(""+returnData);

MP.send(reply, TM);

}

catch(JMSException JMSE)

{

System.out.println("JMS Exception: "+JMSE);

}

}

public static void main(String args[])

{

new Server();

}

@Override

public void onMessage(Message message) {

TextMessage TM = (TextMessage)message;

String reply = null;

String[] input = null;

try

{

if( TM.getText().startsWith("Status"))

{

System.out.println("inside  server");

input = TM.getText().split(",");

reply=status.getInfo(Integer.parseInt(input[1]));

System.out.println("reply in server "+reply);

sendReply(message, reply);

}

}

catch(JMSException JMSE)

{

System.out.println("JMS Exception: "+JMSE);

}

}

public Server()

{

try

{

Properties properties = new Properties();

    properties.put(Context.INITIAL\_CONTEXT\_FACTORY, "org.jnp.interfaces.NamingContextFactory");

    properties.put(Context.URL\_PKG\_PREFIXES, "org.jnp.interfaces");

    properties.put(Context.PROVIDER\_URL, "localhost");

  InitialContext jndi = new InitialContext(properties);

ConnectionFactory conFactory = (ConnectionFactory)jndi.lookup("XAConnectionFactory");

connection = conFactory.createConnection();

session = connection.createSession( false, Session.AUTO\_ACKNOWLEDGE );

StatusQueue = (Queue)jndi.lookup("queue/StatusQueue");

if(null==StatusQueue)

{

StatusQueue = session.createQueue("StatusQueue");

jndi.bind("StatusQueue", StatusQueue);

}

consumer=session.createConsumer(StatusQueue);

consumer.setMessageListener(this);

System.out.println("Server started and waiting for client requests");

connection.start();

}

catch(NamingException NE)

{

System.out.println("Naming Exception: "+NE);

}

catch(JMSException JMSE)

{

System.out.println("JMS Exception: "+JMSE);

JMSE.printStackTrace();

}

}

}

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class FlightStatus {

public static void main(String[] args) {

// TODO Auto-generated method stub

FlightStatus fs=new FlightStatus();

System.out.println(fs.getInfo(1));

}

public String getInfo(int flightId){

Connection con=getConnection();

PreparedStatement preparestatement=null;

ResultSet rs=null;

String info=null;

try

{

preparestatement = con.prepareStatement("select Depart\_date,Flight\_Depart\_Time From airline.flightdetails where Flight\_id=?");

preparestatement.setInt(1, flightId);

rs=preparestatement.executeQuery();

while(rs.next())

{

info="Flight is Departing on "+rs.getString("Depart\_date")+" at "+rs.getString("Flight\_Depart\_Time");

//System.out.println(rs.getString("Depart\_date"));

//System.out.println(rs.getString("Flight\_Depart\_Time"));

}

}

catch (SQLException e)

{

e.printStackTrace();

}

return info;

}

private Connection getConnection() {

Connection con = null;

try

{

Class.forName("com.mysql.jdbc.Driver");

try

{

con = DriverManager.getConnection("jdbc:mysql://localhost:3306/marketplace", "root","install");

}

catch (SQLException e)

{

e.printStackTrace();

}

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

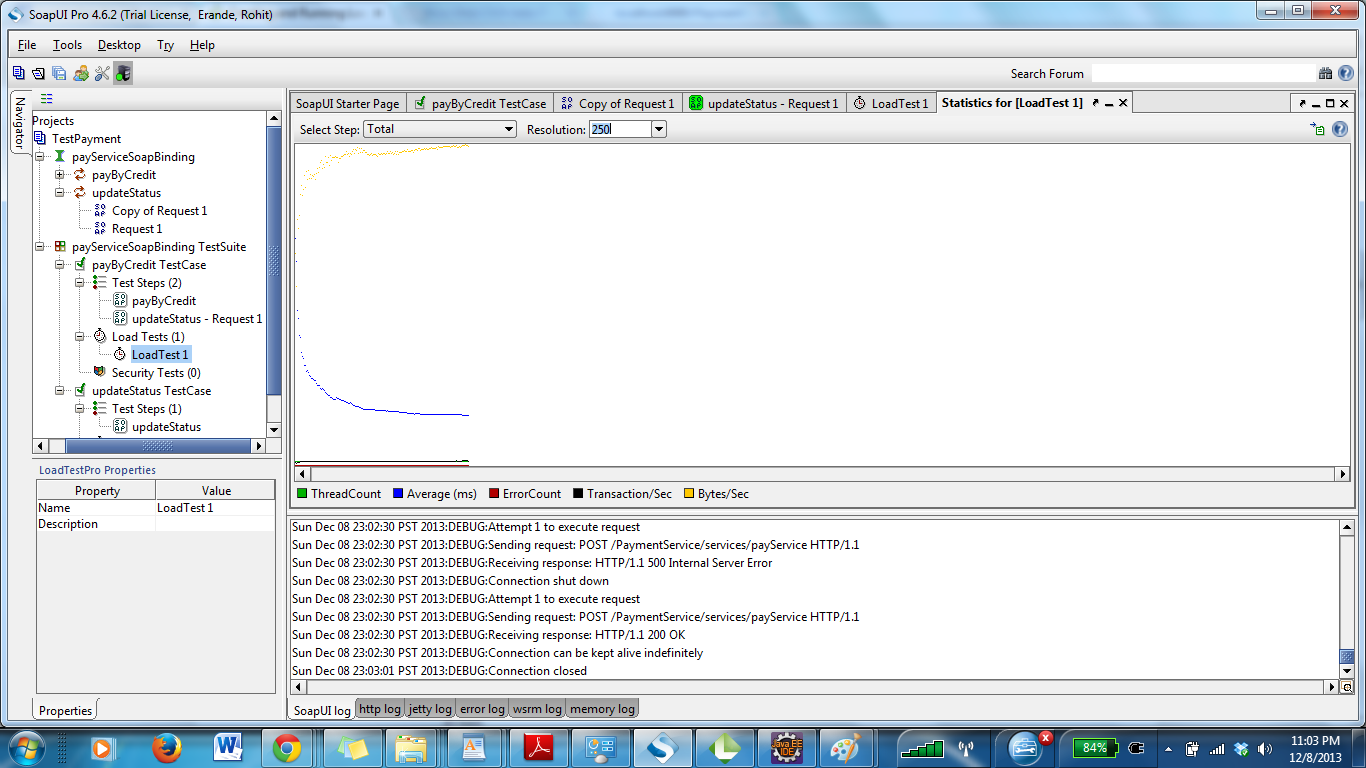
}

return con;

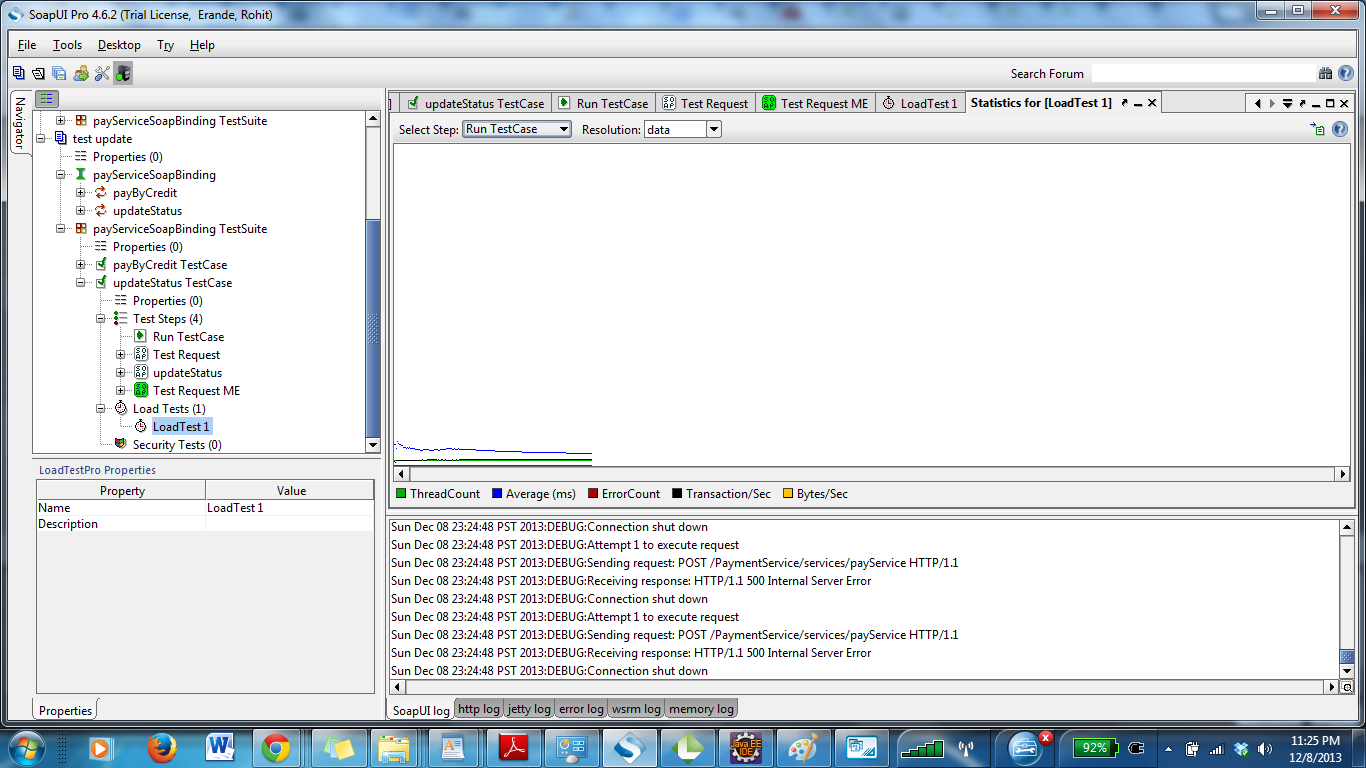
}

}

**Performance measurement**

**Without Caching**

**With Caching**

****

**Observations and lessons learned**

1. It is important to finalize the database design well in advance. Though we managed to discuss the design, we ended up making changes in the way of adding or deleting fields and figuring out referential integrity issues which consumed quite a lot of time.
2. Integrating modules is another area where being in sync with all the team members is very important. This was a major learning in terms of working together as a team to avoid major issues when modules are integrated.
3. Testing along side implementing the code is a good learning in order to spend less time on fixing issues at the end. This saved a lot of time as each of us tested our modules individually as well as end-to-end testing was done upon integration to verify major functionalities.
4. Version control is important for code management. We used emails to forward our code to each other. Using some version control tool would have been a much easier way to handle this. It is a learning to be implemented in the next or upcoming projects and semesters.