Hyresföreningen

Systemutveckling Java & Javaplattformen

Individuellt Projekt SQL/JSP/Servlets av Stefan Sundström

Bakgrund

Applikation används för att administrera ett litet bostadsföretag - en databas håller register över administratörer, hyresgäster, lägenheter och hus. Administratörerna kan lägga till, ta bort och uppdatera hyresgäster samt skriva ut detaljer och historik kring både byggnader och hyresgäster. Administratörerna på företaget har egna användare och lösenord som verifieras genom en inloggning.

Syfte

Förutom att uppfylla uppgiftens krav och visa på praktisk användning av SQL och Java så är syftet att få en bra förståelse för JSP, Servlets och designmönstret MVC. Servleten står mellan användarens gränssnitt och kommunikationen med databasen för att säkerhetsställa att användaren inte kommer i kontakt med logistik och uppgifter kring hur systemet är uppbyggt.

För att kunna köra applikationen

För att kunna köra programmet så måste en databas, några tabeller och en användare skapas. Texten i filen ”SQL-data” kan kopieras och köras i MySQL-WorkBench vilket skapar all data som behövs i databasen. Det finns två krypterade användare med lösenord som kan användas vid inloggningen.

User: Holger Login: Holger eller User: Stefan Login: Stefan.

Dessa tre JAR-filer måste också ligga i WEB-INF. En är för JDBC/uppkopplingen och de andra för att kunna köra JSTL-taggar på JSP-sidorna.

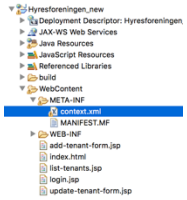
Om programmet körs från en IDE så körs **Login.java** för att starta programmet, man kan också klistra in url’en i en webbläsare och då räcker det med ”http://localhost:8080/Hyresforeningen”

DataSource för en skapa en pool av uppkopplingar

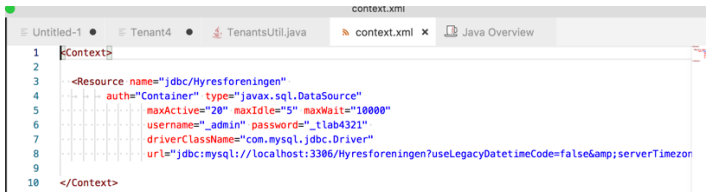
För att applikation ska kunna kopplas mot databasen och användas och av flera användare samtidigt så används en så kallas ”*Connection Pool*”. När en uppkoppling är använd så lägger den sig och väntar i en pool på en ny användare.

Först och främst behövs JDBC drivern (JAR) som placeras i ”WEB_INF/lib” - sedan måste poolen konfigureras, för detta används en config-fil (context.xml).

Context.xml i Eclipse



Context-xml i kod-editerare



Contextfilen talar om för TomCat hur uppkoppling mot databasen ska konfigureras, namnet längst uppe i koden är namnet som man refererar till från koden, ”auth: Container” betyder att TomCat-servern sköter autentiseringen, ”type=javax.sql.DataSource” är namnet på javainterface som används för att kommunicera med poolen, termen ”DataSource” kan likställas med ”Connection Pool”. Man sätter sedan ”storleken” på datapoolen, Username och password är det som är kopplat till den användare man satt i SQL-databasen, url är den URL som används vd uppkopplingen mot databasen (Driver/mySql/EgnaDatorn/Port/Database/Tidszonfix.)

I Servletklassen så anges en referens till datapoolen - @Resource talar alltså om att det är Context-filen med namnet ”jdbc/Hyresforeningen” som ska användas av Tomcat.

Login

Det första som händer är att man får skriva in ett användarnamn och ett lösenord, dessa hämtas av Servleten och skickas vidare till DBUtility som skickar ”välj de rader i databasen som innehåller lösenordet och användarnamnet” till _admin-tabellen”. Genom metoden rs.next() så får man veta om det finns en rad som överensstämmer och detta returneras sedan som från metoden som en boolean. Stämmer lösenord/användare så returneras alltså ”true”. Lösenordet är krypterat och anledningen är egentligen mest för att jag störde mig på att databasen inte gör skillnad på små och stora bokstäver. Krypteringen löser detta (men lösenordet skickas ändå POST/okrypterat).

```
public boolean validateAdmin(String _username, String _password)
{

        _password = PassCrypt.hashPassword(_password);
        _username = PassCrypt.hashPassword(_username);


    boolean status=false;

    Connection conn = null;
    String query = "select * from _admin where _username =? and _password =?";


    try {
        conn = dataSource.getConnection();
        PreparedStatement pst = conn.prepareStatement(query);
        pst.setString(1, _username);
        pst.setString(2, _password);
        ResultSet rs = pst.executeQuery();

        status = rs.next();


        }

    catch(Exception e){System.out.println(e);}

    return status;
    }
```

Krypteringen görs med hjälp av klassen MessageDigest, ett objekt skapas utifrån algoritmen SHA-256. En valfri sträng ”SALT” skapas, denna sträng och det lösenord användaren matar in görs om till en sekvens av bytes, läggs ihop i MessageDigest-objektet där sedan metoden digest kan skapa ett fält av bytes (ex.100, 113, -103, 104, 16, 60.. osv). Detta fält sätts sedan som inparameter i metoden ”bytesToHex” som skapar ett nytt fält med dubbelt så många index som gick in i metoden, vad som händer i for-loopen har jag inte satt mig in men detalj men det ser ut som varje index modifieras utifrån hexArray-fältet - sedan returneras char-sekvensen som en sträng. Detta är det krypterade lösenordet som sedan kan lagras i databasen.

```
import java.security.MessageDigest;

public class PassCrypt {


    final protected static char[] hexArray = "0123456789ABCDEF"
            .toCharArray();

      public static String bytesToHex(byte[] bytes) {
        char[] hexChars = new char[bytes.length * 2];
        int v;
        for (int j = 0; j < bytes.length; j++) {
          v = bytes[j] & 0xFF;
          hexChars[j * 2] = hexArray[v >>> 4];
          hexChars[j * 2 + 1] = hexArray[v & 0x0F];
        }
        return new String(hexChars);
      }

      // Change this to something else.
      private static String SALT = "123456";

      // A password hashing method.
      public static String hashPassword(String in) {
        try {
          MessageDigest md = MessageDigest
              .getInstance("SHA-256");
          md.update(SALT.getBytes());
          md.update(in.getBytes());


          byte[] out = md.digest();
          return bytesToHex(out);
        } catch (NoSuchAlgorithmException e) {
          e.printStackTrace();
        }
        return "";
      }

}
```

Servlet

En Servlet ligger mellan gränssnittet och databas-kopplingen, användaren kommer på så sätt inte i kontakt med metoder som är direkt kopplade till databasen och ser inga detaljer kring hur systemet är byggt. Servleten kommunicerar med JSP-sidan med hjälp av så kallade HttpServletRequest och HttpServletResponse.

Servleten ärver av klassen ”HttpServlet” och i klassen skrivs en init-metod över, den fungerar som en konstruktor och skapar ett objekt av klassen DBUtility med en uppkoppling som inparameter.

```
@Override
public void init() throws ServletException

{
    super.init();
    try {dBUtility = new DBUtility(dataSource);}
    catch (Exception e) {throw new ServletException(e);}
}
```

När applikationen körs så hämtas ett kommando från JSP-sidan som sätter switchsatsens case och avgör vilken metod som ska anropas.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    try {
        _command = request.getParameter("command");

        if (_command == null) {_command = "VALIDATE";}

        switch (_command)
        {


        case "VALIDATE":

            if (Validmess == false)
            {
                listEmptyA(request, response);
                _command="LOAD_EMPTY_A";
            }
            else {doPost(request, response);}
            break;
        case "LIST":
            listTenants(request, response);
            break;

        case "LIST_EMPTY_A":
            listEmptyA(request, response);
            break;

        case "LIST_A":
            listA(request, response);
            break;

        case "ADD" :
            addTenant(request, response);
            break;

        case "ADD_FROM_ID" :
            addTenant_from_id(request, response);
            break;

        case "LOAD" :
            loadTenant(request, response);
            break;

        case "LOAD_A" :
            loadA(request, response);
            break;

        case "UPDATE" :
            updateTenant(request, response);
            break;

        case "UPDATE_APARTMENT" :
            updateApartment(request, response);
            break;

        case "INFO_APARTMENT" :
            infoApartment(request, response);
            break;

        case "DELETE" :
            deleteTenant(request, response);
            break;


            default:

                addTenant_from_id(request, response);

        }

        listTenants(request, response);
    } catch (Exception e) {

        e.printStackTrace();
    }

}
```

Man kan hämta värdet till Servleten från JSP-sidan genom metoden getParameter(). Här nedan hämtas ”command” som styr switchsatsen.

Servlet

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    try {
        _command = request.getParameter("command");

        if (_command == null) {_command = "VALIDATE";}

        switch (_command)
        {

        case "VALIDATE":
```

Värdet som man plockar in i metoden är beror alltså på namnet som sätts på värdet på JSP-sidan. Ex. Värdet ”LOAD” hämtas genom request.getParameter(”command”).

JSP-sida

```
<c:param name="command" value="LOAD"/>
<c:param name="tenantID" value="${tempTenant.id}"/>
```

Objekt skapas

För att lätt kunna jobba med att ”flytta kopior” av de tabellvärden man hämtar från databasen till JSP-sidan så används ärvande klasser som representerar tabellerna.

```
public class Tenant extends Apartment
{
int id;
int apartmentNumber;
String firstName;
String lastName;
String ss_number;
String mobile;
String email;
String _from;
String _until;
String notes;
```

```
public class Apartment extends House
{

    int id;
    int house_number;
    double size;
    int rooms;
    boolean balcony;
    int floor;
    double bofond;
    double rent;
    String fridge;
    String freezer;
    String stove;
    String a_notes;
```

```
public class House
{

    int id;
    boolean elevator;
    boolean gym;
    boolean sauna;
    boolean storage_room;
    String construction_date;
    String address;
    String postal_code;
    String city;
```

Dessa klasser har konstruktörer som fylls med motsvarande tabellkolumnvärden samt *getters* som kan användas för att hämta dessa värden. JSTL-taggarna på JSP-sidorna ersätter så kallade scriplets och säger egentligen t.ex. tempTenant.getfirstName och anropar en getter-metod i klassen Tenant.

JSP

```
<td>${tempTenant.firstName}</td>
<td>${tempTenant.lastName}</td>
<td>${tempTenant.address}</td>
<td>${tempTenant.postal_code}</td>
<td>${tempTenant.city}</td>

<td>${tempTenant.mobile}</td>
```

För att JSP-sidorna ska kunna köra dessa måste en tagg läggas till i toppen av JSP-dokumentet.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Lägga till en hyresgäst

I metoden nedan så hämtas alla input-boxars värden genom metoden getParameter och dessa värden sparas i nya variabler som används för att skapa ett objekt som motsvarar de hyresgäster som finns lagrade i databasens tabell. Sedan körs körs dbUtility-metoden addTenant och skickar med objektet som in-parameter.

Servlet

```
private void addTenant(HttpServletRequest request, HttpServletResponse response) throws Exception
    {

        int apartmentNumber = Integer.parseInt(request.getParameter("apartmentNumber"));
        String firstName = request.getParameter("firstName");
        String lastName= request.getParameter("lastName");
        String ss_number= request.getParameter("ss_number");
        String mobile= request.getParameter("mobile");
        String email= request.getParameter("email");
        String _from= request.getParameter("_from");
        String _until= request.getParameter("_until");
        String notes= request.getParameter("notes");


        Tenant tempTenant = new Tenant(apartmentNumber, firstName, lastName, ss_number,mobile, email, _from,_until, notes);
        dBUtility.addTenant(tempTenant);

        listTenants(request, response);
    }
```

I addTenant så skapas en uppkoppling mot databasen, en query talar om att en ny post i tabellen ”Tenant” ska skapas med värden som hämtas från objektet som skickats in i metoden.

DBUtility

```
public void addTenant(Tenant tempTenant) throws SQLException

    {

        Connection conn = null;
        PreparedStatement pstmt= null;
        String query ="";

        try {
            conn = dataSource.getConnection();
            query = "insert into Tenant (apartmentNumber, firstName, lastName, "
                    + "ss_number,mobile, email, _from,_until, notes) "
                    + "values(?,?,?,?,?,?,?,?,?)";
            pstmt = conn.prepareStatement(query);
            pstmt.setInt(1, tempTenant.getApartmentNumber());
            pstmt.setString(2, tempTenant.getFirstName());
            pstmt.setString(3, tempTenant.getLastName());
            pstmt.setString(4, tempTenant.getSs_number());
            pstmt.setString(5, tempTenant.getMobile());
            pstmt.setString(6, tempTenant.getEmail());
            pstmt.setString(7, tempTenant.get_from());
            pstmt.setString(8, tempTenant.get_until());
            pstmt.setString(9, tempTenant.getNotes());
            pstmt.execute();

        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        finally {conn.close(); pstmt.close(); conn.close();}


    }
```

Hämta specifika objekt från ID

I många fall så måste ett specifikt objekts ID hämtas från en post i tabellen för att sedan användas för att hämta data till ett formulär som sedan ska kunna uppdateras. I bilden ovan så markeras ”hyr ut” med muspekaren på översta lägenheten i tabellen och man ser då URL’en längst nere till vänster som plockar med sig command ”ADD_FROM_ID” samt emptyA_ID=1. Varje post skickar alltså med sitt specifika ID via GET i Url’en.

Servlet

```
private void addTenant_from_id(HttpServletRequest request, HttpServletResponse response) throws Exception
{

    int emptyA_ID = Integer.parseInt(request.getParameter("emptyA_ID"));
    LocalDateTime date = LocalDateTime.now();
    String _today = date.toString();
    String today = _today.substring(0, 10);

    request.setAttribute("THE_APARTMENT", emptyA_ID);
    request.setAttribute("DATE", today);

    RequestDispatcher dispatcher = request.getRequestDispatcher("/add-tenant-form_from_ID.jsp");
    dispatcher.forward(request, response);
    listTenants(request, response);
}
```

Lägenhetsnumret hämtas av Servleten, LocalDateTime används för att få dagens datum och tid där substring() körs för att få enbart datumet i samma format som SQL-databasens datum. Attributen på requesten får namn och värden sätts till lägenhets-id och datumet. RequestDispatcher-objektet skapas genom att metoden getRequestDispatcher tar in JSP-sidans ”sökväg” och sedan kan request och respons skickas för att utbyta data.

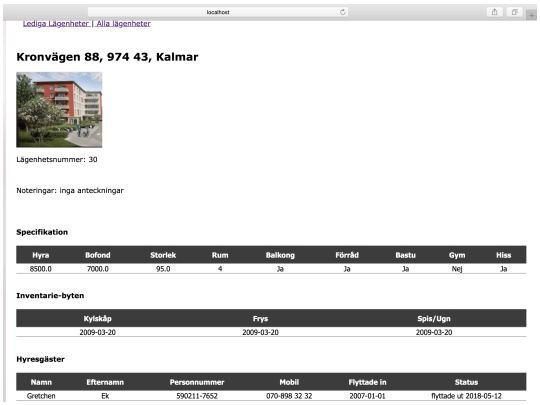
JSP

```
<form action ="Servlet" method="GET">
<input type="hidden" name="command" value="ADD"/>
<input type="hidden" name="Empty_A_ID" value="${THE_APARTMENT}"/>
<input type="hidden" name="todays_date" value="${DATE}"/>

<table>
<tbody>
<tr>

<tr><td><label>Lägenhetsnummer</label></td>
<td><input type="text" name="apartmentNumber" value ="${THE_APARTMENT}"/></td></tr>
<tr><td><label>Förnamn</label></td>
<td><input type="text" name="firstName"></td></tr>
<tr><td><label>Efternamn</label></td>
<td><input type="text" name="lastName"></td></tr>
<tr><td><label>Personnummer</label></td>
<td><input type="text" name="ss_number"></td></tr>
<tr><td><label>Mobil</label></td>
<td><input type="text" name="mobile"></td></tr>
<tr><td><label>Email</label></td>
<td><input type="text" name="email"></td></tr>
<tr><td><label>Inflyttad</label></td>
<td><input type="text" name="_from" value = "${DATE}"></td></tr>
<tr><td><label>Noteringar</label></td>
<td><input type="text" name="notes"></td></tr>
<tr>
<td><label></label></td>
<td><input type="submit" value="Spara" /></td>
</tr>

</tbody>
```

JSP-sidan öppnas och apartmentNumber får värdet `"${THE_APARTMENT}"` som sattes som Attribut på requesten i Servleten, samma gäller datumet. När fälten är ifyllda och

användaren trycker spara så anropas metoden doGet(switchsatsen), skickar med command ”ADD”.

SQL



Den mest tidskrävande SQL-satsen är den som skriver ut info om lägenheterna. Metoden tar in lägenhetsnumret, deklarerar ett fält med hyresgäster (hyresgästen är subklassen och all information behövs). SQL-satsen säger ”välj alla detaljer kring hyresgäst/lägenhet/hus där lägenhetsnumret är ?”. inparametern/lägenhetsnummret sätts som värde på ”?” och då kan alla poster som har en hyresgäst knuten till lägenhetsnumret hämtas. En while-sats går igenom alla lägenheter och skapar ett hyresgästobjekt (subklassen) som tar emot alla kolumnvärden i sin konstruktor, objektet läggs till fältet med hyresgäster och när ResultSetet har gåtts igenom så returnerar metoden en lista som motsvarar alla hyresgäster som bott eller bor i lägenheten.

```
public List<Tenant> getApartmetInfo(String apartmentID) throws Exception
{
    List<Tenant> apartmentInfoList = new ArrayList<>();

    Tenant theTenant = null;
    Connection conn = null;
    PreparedStatement statement = null;
    ResultSet rs = null;
    int _apartmentID;


    try
    {

        _apartmentID = Integer.parseInt(apartmentID);
        conn = dataSource.getConnection();

        String sql ="select h.elevator, h.gym, h.sauna, h.storage_room, h.construction_date,h.address,"
                + "h.postal_code, h.city, a.id, a.house_number, a.size, a.rooms, a.balcony, a.floor, a.bofond,"
                + "a.rent, a.fridge, a.freezer, a.stove, a.notes, t.id, t.apartmentNumber, t.firstName, t.lastName, "
                + "t.ss_number, t.mobile, t.email, t._from, t._until,t.notes from apartment as a left join tenant as t on "
                + "t.apartmentNumber = a.id left join house as h on h.id=a.house_number where a.id = ?";


        statement = conn.prepareStatement(sql);
        statement.setInt(1, _apartmentID);
        rs = statement.executeQuery();

        while (rs.next())

    {
            boolean elevator = rs.getBoolean("h.elevator");
            boolean gym = rs.getBoolean("h.gym");
            boolean sauna = rs.getBoolean("h.sauna");
            boolean storage_room = rs.getBoolean("h.storage_room");
            String construction_date = rs.getString("h.construction_date");
            String address= rs.getString("h.address");
            String postal_code= rs.getString("h.postal_code");
            String city= rs.getString("h.city");
            int id = rs.getInt("a.id");
            int house_number = rs.getInt("a.house_number");
            double size = rs.getDouble("a.size");
            int rooms= rs.getInt("a.rooms");
            boolean balcony = rs.getBoolean("a.balcony");
            int floor= rs.getInt("a.floor");
            double bofond = rs.getDouble("a.bofond");
            double rent = rs.getDouble("a.rent");
            String fridge= rs.getString("a.fridge");
            String freezer= rs.getString("a.freezer");
            String stove= rs.getString("a.stove");
            String a_notes= rs.getString("a.notes");
            int id2 = rs.getInt("t.id");
            int apartmentNumber = rs.getInt("t.apartmentNumber");
            String firstName = rs.getString("t.firstName");
            String lastName = rs.getString("t.lastName");
            String ss_number = rs.getString("t.ss_number");
            String mobile = rs.getString("t.mobile");
            String email = rs.getString("t.email");
            String _from = rs.getString("t._from");
            String _until = rs.getString("t._until");
            String notes = rs.getString("t.notes");


        theTenant = new Tenant(elevator, gym, sauna, storage_room, construction_date,address,postal_code, city, id,
                house_number,size, rooms, balcony,floor, bofond, rent, fridge, freezer, stove, a_notes, id2, apartmentNumber,
                firstName, lastName,ss_number, mobile, email,_from, _until, notes );

        apartmentInfoList.add(theTenant);

    }




    return apartmentInfoList;
    }

    finally {conn.close(); statement.close(); rs.close();


    }
```

Svårigheter med Projektet

Det mest tidskrävande har varit ren felsökning, till en början var det också svårt att förstå exakt hur kommunikationen mellan JSP/Servlet/databas fungerar. Att bygga en MVC-modell komplicerar utskriften av databas-inormationen väldigt mycket då objekt/fält måste skickas istället för att bara skriva ut från ett ResultSet direkt på gränssnittet.

Ett mindre problem var historiken, till en början tänkte jag att curdate() i en databas alltid var dagens datum men upptäckte att det förstås bara lagrade just den dagens datum. Därför blev kunde jag inte bara säga att alla som bor i lägenheten curdate() är nuvarande hyresgäster. Det fungerar bara samma dag som man fyller på tabell-data.

Lösningen på det blev att säga att låta utflyttningsdatumet vara null- (WHERE t_until IS NULL)

```
String query = "select t.id, t.firstname, t.lastname, h.address, h.postal_code, h.city,"
        + " t.mobile from apartment as a join Tenant as t  "
        + "on t.apartmentNumber = a.id join House as h on h.id = a.house_number "
        + "where t._until  IS NULL  order by t.id desc";
```

Ett annat problem vara att jag läste planeringen slarvigt och tänkte att jag hade en vecka till på mig som jag hade tänkt lägga på css/gränssnitt. Jag tycker det funkar som det är men hade förstås kunnat se roligare ut.

Saker jag hade kunnat göra bättre

Jag hade som sagt gärna hade jobbat lite med gränssnittet- haft en logotype och varit tydligare med att det är ”Hyresföreningens-Administrationsprogram” istället för att bara speca funktionerna när man väl använder programmet.

Jag lämnade också undantagshanteringen på något ställe, man hade kunnat kontrollera så att en email-adress innehåller ”@” och i ett formulär kan man krascha programmet om man skriver in en bokstav på lägenhetsnumret. Jag är ändå rätt nöjd då man kan röra sig i programmet utan buggar och om man inte är ute efter att krascha det så fungerar det bra.

Jag hade också egentligen velat lägga upp applikationen på en riktig server men tiden tog slut.

Krypteringen av lösenord hade jag gärna lagt en dag till på och bilderna som används på info-sidan hade jag velat lägga som länkar i databasen och inte placera bland front-end filerna.

Sedan hade det varit bra att skriva in en funktion i Servlet-klassen som kontrollerar och visar vem som är inloggad samt möjligheten att sortera tabellerna efter valfri variabel från JSP-sidan.

Planen var att använde Maven istället för att manuellt importera JAR-filerna men även det blev att prioritera bort.

Sammanfattningsvis

Redan innan jag började med att programmera så har jag alltid velat göra en hemsida med Java, jag har skrivit lite HTML/CSS förut och har funderat kring hur man använder JAVA-kod mot HTML/JSP men det har sett så komplicerat ut. Nu har jag faktiskt tagit mig igenom ett projekt där jag fått ganska bra förståelse för hur det fungerar. Rent tidsmässigt så trodde jag nog att det skulle vara lättare/gå snabbare.

Jag tycker själv att jag har fått en bättre förståelse för både SQL, hur allt är sammankopplat och ren objektorienterad programmering.