HW 5 – Database: Properties & Patients

Contents

[1 Overview 1](#_Toc130989342)

[2 Requirement 1 – Default.aspx 2](#_Toc130989343)

[3 Requirement 2 – Patients1.aspx 3](#_Toc130989344)

[4 Requirement 3 – Patients2.aspx 4](#_Toc130989345)

[5 Requirement 4 – Patients3.aspx 5](#_Toc130989346)

[6 Submission 6](#_Toc130989347)

[7 Grading Criteria 6](#_Toc130989348)

[8 Notes & Global Hints 6](#_Toc130989349)

# Overview

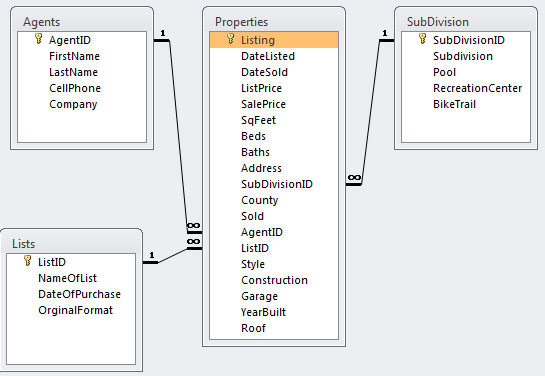
This is an individual assignment. You will develop four (independent) pages which interact with a database. You have been provided the UI for each of these four pages. The average time to complete HW 11 is 12 hours. Most people took 10-15 hours, some, up to 20. Some people did only portions of it to receive a B or C grade, and they spent 6-10 hours.

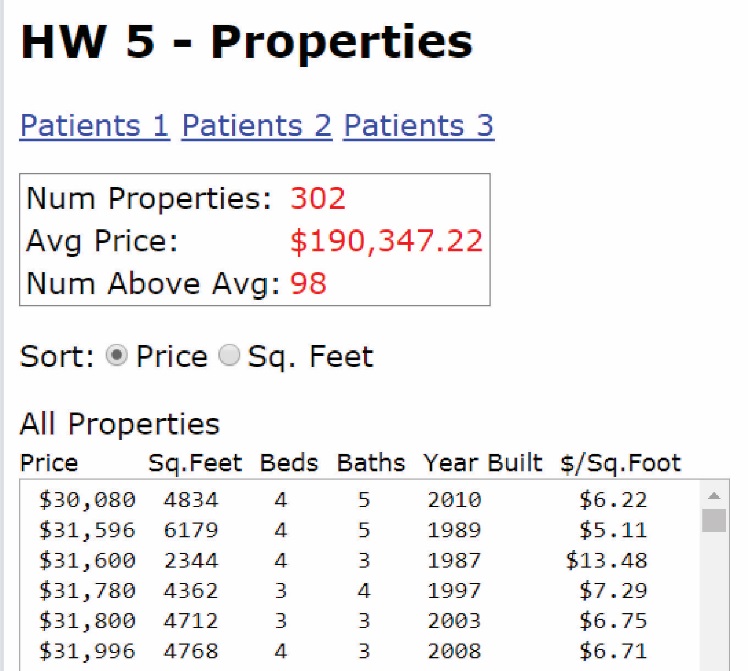
To get started, do the following:

1. Download *hw05\_shell.zip* from the Schedule and unzip on your drive.
2. The website will be found in a folder named: *hw5\_shell.* I do not recommend renaming this unless you know what you are doing!
3. Open the website in VS (double-click, *hw5\_Shell.sln*)

Detailed requirements are on the following pages. Once reading through Requirements 1-4, you should read [Notes & Global Hints](#_Notes_&_Global) before beginning to code.

# Requirement 1 – Default.aspx

1. For this page (*Default.aspx*) you will use the *Properties* table in the *properties.mdb* database as shown on the right.

When complete, *Default.aspx* will look as shown on the right. Requirements:

1. When the page is first displayed the three statistics (shown in red on GUI) are calculated and shown by reading all data from the *Properties* table. Notes:
2. You will only be using these fields: *ListPrice*, *SqFeet*, *Beds*, *Baths*, *YearBuilt*.
3. You are required to create a *Property* class to hold the data once it is read in, and put into a *List<Property>*.

1. The third statistic is the number of properties that have a list price above the average list price.

Hint: You’ll need to go through the list of properties once to find the number of properties and the average price. Then, you’ll need to go through it again to find the number that are above the average price.

1. Also, when the page is first displayed, the data for all properties is shown sorted by price. The data does not need to be as neatly formatted as above, but should not be too messy. Note that the last column, *“$/Sq.Foot”* is calculated, *e.g. Price/Sq.Feet*.

Hints:

* The *Property* class should also have a field for *Price/Sq.Feet*. The constructor can simply calculate it.
* This statement will produce a line of text that is nicely formatted:

String prop = String.Format("{0,8:$0,0} {1,5:0} {2,2:0} {3,2:0} {4,4:0} {5,6:$0.00}", p.ListPrice, p.SqFeet, p.Beds, p.Baths, p.YearBuilt, p.PricePerSqFoot);

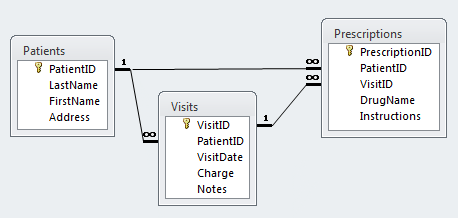
1. When the radio button is pressed the data is immediately sorted on either Price or Sq. Feet.

Hints:

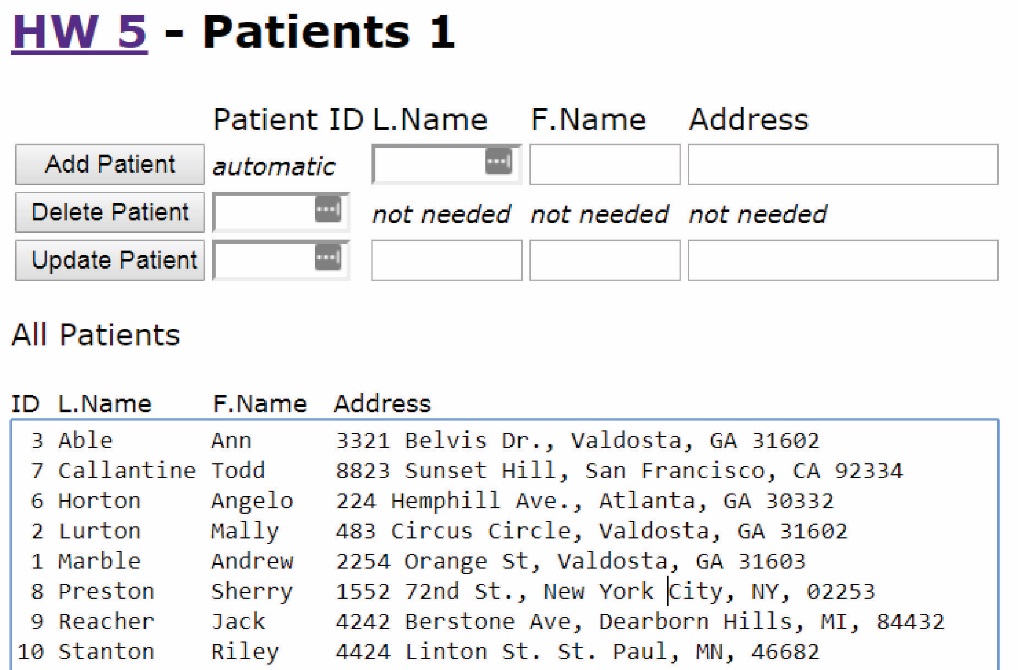
* Set *AutoPostback* to *true* for the *RadioButtonList* (it may be already)
* You need two slightly different SQL statements, the ORDER BY clause is the only part that is different.

# Requirement 2 – Patients1.aspx

1. For Requirements 2-4, you will use the *Patients* database:



1. When complete, *Patients1.aspx* will look as shown below.



1. When the page is first displayed, it shows all the patients.

Hints:

* Write a method for this, *displayPatients* and call it in *Page\_Load* the first time the page is accessed.
* This statement will produce a line of text that is nicely formatted:

String patient = String.Format("{0,2:0} {1,-10:0} {2,-8:0} {3,-40:0}", id, lname, fname, address);

1. The user can add, delete, and update a patient by supplying the fields shown and the display showing all patients is automatically updated.

Hint: after performing an add, delete, or update, call your *displayPatients* method.

1. You are required to use *parameterized queries*.

Hint: see the *Database Programming Primer* notes, items 16-19 (Parameterized Queries).

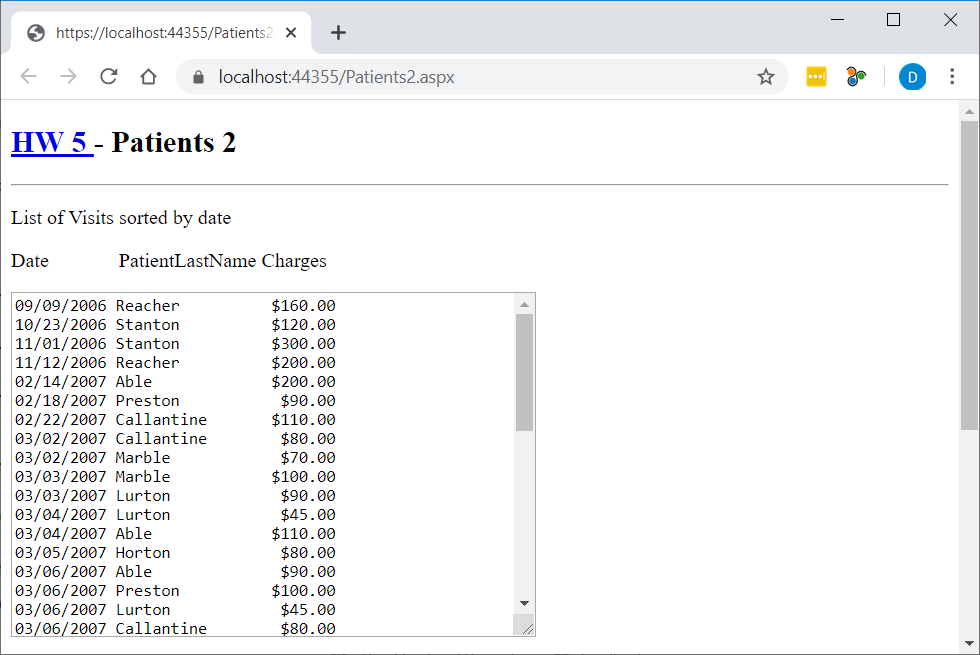
1. The page should not crash if you try to delete a patient that has visits associated with it.

Hints:

* To handle the case where a patient has visits and the user attempts to delete, see the *Database Programming Primer* notes, item 22.
* To test a successful delete, just add a patient, and then delete it and this will work because the patient doesn’t have any visits.

# Requirement 3 – Patients2.aspx

1. When complete, *Patients2.aspx* will look similar to what is shown below.



1. When the page is loaded it displays all visit dates, the corresponding patient last name, and the charges for the visit.

Hints:

* You will need an inner join to accomplish this.
* For the data reader to read Visit.VistDate from the data reader (assuming it is in column 1)

DateTime dtDate = (DateTime)dr.GetValue(0);

And then turn into a string:

string date = dtDate.ToString("MM/dd/yyyy");

* To read Visit.Charge from the data reader (assuming it is in column 3) (not sure why you need to read it as decimal, double will not work. The database field itself is “Currency”)

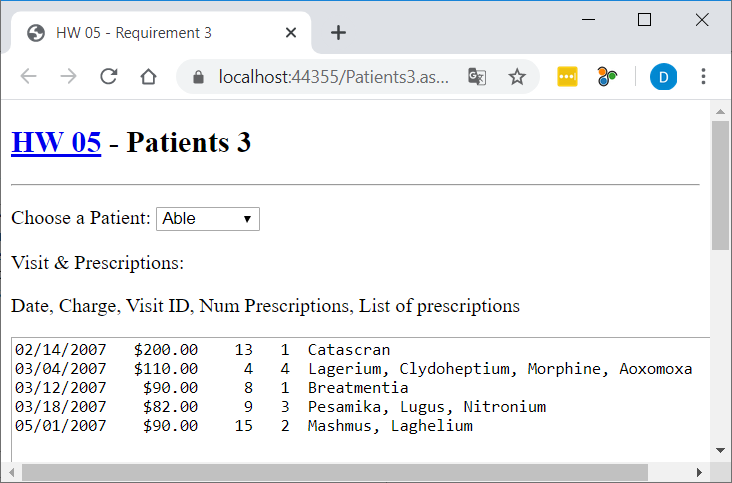
decimal charge = dr.GetDecimal(2);

* This will format the output nicely

string visit = String.Format("{0,10:0} {1,-14:0} {2,9:$0,0.00}", date, lname, charge);

# Requirement 4 – Patients3.aspx

1. When complete, *Patients3.aspx* will look similar to what is shown below.



1. When the page loads, populate the drop down with the last name (“LName”) of all patients. Use the “PatientID” for the Value.
2. When a patient is selected, do one of two things:
3. (easier) Display the date, charge, and visit id for all the patient’s visits, sorted by date. See Example 5 in the code download. Actually, it is simpler than that, you don’t even need an inner join: just use the PatientID to select from the visits table.
4. (harder) Display the date, charge, number of prescriptions for the visit, visit id, and a list of all the prescriptions for the visit.

Hints (for harder version):

* Here is an algorithm for *SelectedIndexChanged* (which should be modified because it will be very long)

Use the selected PatientID to get the visits (VisitID, VisitDate, Charge) – no inner join is needed

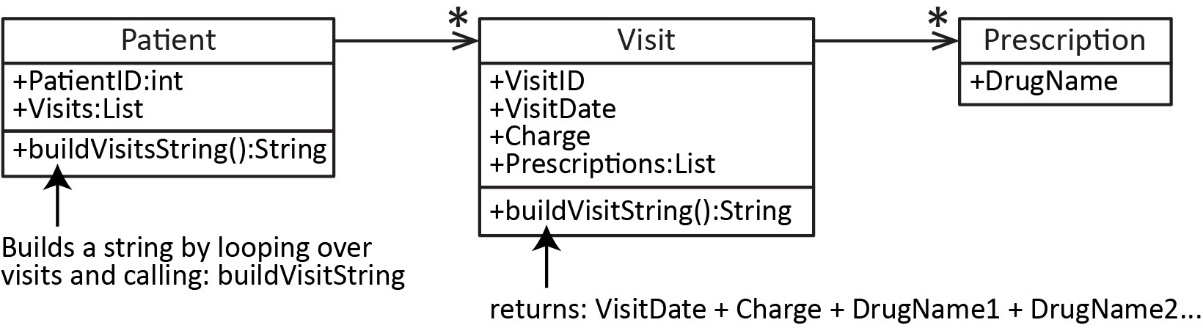
Loop through the results (visits), at each iteration grab the next VisitID

Use VisitID to get prescriptions (DrugName) – no inner join is needed

Associate current visit with prescriptions\*

Build display message.

\*The harder part is making this association. I looked back and I did NOT require you to use OO for this problem/requirement. However, I would recommend doing so. You have to do more work up front, but then the management of the relationship between a particular visit and its prescriptions is easier (and easier to debug and maintain). Thus, I would recommend this structure:



* The format of the display can simply have each value separated by a comma for simplicity. Or, you can modify one of the String.format hints I provided several places earlier to make it look as above.

# Submission

1. Zip your **solution folder** into a file named: *hw05\_shell.zip*. The following must be true:
2. When I unzip, it unzips to a folder named: *hw05\_shell*
3. There is a *hw05\_Shell.sln* file in the solution folder that when double-clicked launces your project in VS 2019.
4. Submit on Blazeview in the HW 5 dropbox.

# Grading Criteria

|  |  |  |  |
| --- | --- | --- | --- |
| **Req** | **Descripton** | **Points** |  |
| 1 | Num Properties correct | 5 |  |
|  | Avg Price correct | 8 |  |
|  | Num Above Avg correct | 8 |  |
| 2 | All properties displayed | 8 |  |
|  | $/Sq. Foot displayed | 8 |  |
| 3 | Sort works | 8 |  |
| 4 | Add works | 9 |  |
|  | Delete works | 9 |  |
|  | Update works | 9 |  |
| 5 | Use parameterized queries | 5 |  |
| 6 | Visits and patients displayed | 5 |  |
| 7 | Dropdown display patients | 9 |  |
| 8 | Visits displayed for selected patient | 9 |  |
|  | **or** Visits & num prescriptions displayed |  | 4 |
|  | **Total** | **100** | **95** |

# Notes & Global Hints

1. The solution has an *App\_Data* folder with the two databases needed: *properties.mdb* and *patients.mdb.* The *Web.config* file contains

<connectionStrings>

<add name="Access\_Patients"

connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=|DataDirectory|\patients.mdb"

providerName="System.Data.OleDb" />

<add name="Access\_Properties"

connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=|DataDirectory|\property.mdb"

providerName="System.Data.OleDb" />

</connectionStrings>

In the following requirements, if you get an error about the “provider”, then you may need to generate these connection strings yourself so that it uses the provider on your machine. If this is the case, use the connection wizard to create connections to each of the databases, naming the connection string, “Access\_Patients” or “Access\_Properties”. If you need more guidance:

1. Open *Web.config* and delete the *connectionStrings* node shown above.
2. Open Lab 9 and reference Section 4 (Databinding: DataSource & Gridview), all steps, *a-k* with *patients.mdb*. Important: Step *i,* use the “name” from above, “Access\_Patients”. Steps *j* and *k* don’t matter, but just do them anyway.
3. Open *Web.config* and examine the connection string.
4. Repeat Step *b* (immediately above) with *property.mdb*. On step i, use the “name” from above, “Access\_Properties”.
5. Open *Web.config* and examine the connection string.
6. You can delete the data source(s). Doing so will not delete the connection strings.
7. Each page has an instance variable, *dbType* which specifies the database you will be using on that page. For example:

public partial class Default : System.Web.UI.Page {

string dbType = "Access\_Properties";

public partial class Patients1 : System.Web.UI.Page {

string dbType = "Access\_Patients";

*etc.*

1. You have been supplied with a *ConnectionFactory.cs* class that has a static method, *getCommand* method which accepts a string specifying the database you are using. Thus, to use the method, you simply need to pass it the *dbType* instance variable. For example:

try {

IDbCommand cmd = ConnectionFactory.GetCommand(dbType);

cmd.CommandText = getSQL();

cmd.Connection.Open();

IDataReader dr = cmd.ExecuteReader();

txtMsg.Text = "Attempting to read from: Access" + " database" + Environment.NewLine;

txtMsg.Text += "IDbConnection.State: " + cmd.Connection.State.ToString() + Environment.NewLine; ;

txtMsg.Text += "IDataReader.IsClosed: " + dr.IsClosed + Environment.NewLine;

txtMsg.Text += "cmd.CommandText: " + cmd.CommandText + Environment.NewLine + Environment.NewLine;

...

dr.Close();

cmd.Connection.Close();

}

catch (Exception ex) {

txtMsg.Text = "\r\nError in SelectedIndexChanged\r\n";

txtMsg.Text += ex.ToString();

}

I can’t remember if you must *try/catch* in database operations in ASP.NET. However, I strongly recommend that you use a *catch* block similar to above (and the lab) and also add the displays shown in the *try* block above.like you did in the lab. In the *catch* block, you should print an exception that is thrown.

1. Database work is **very** tedious. The first thing to figure out when you get an error is IF your SQL statement is correct. Notice in the *try* block above, we print it out. You should look at it carefully to find your error. It may be hard to see in the textbox on the page, so copy it to word and look at it there. Common errors in a SQL statement:

* Missing a space between two things
* A comma after the last field selected in a Select statement
* Misspelled the name of a field. Remember it must be the exact name in the database.

1. Remember: baby-steps is the key to programming in general, and especially in web programming.
2. I recommend writing helper methods. For example, I have 6 helper methods in my solution for the first page (below), *Default.aspx.*