DatasetTable Mapping Example

Map tables and columns into datasets.

This solutions walks through three different approaches to filling a DataGrid with information.

Download the Solution Example:

Solution file: DatasetTable Mapping Example.dbsln
Database file: DatasetTable Mapping Example.db

Summary

When working with databases, there are often multiple ways to get the data you need. This guide walks through three different approaches to filling a DataGrid with information, each with its own strengths:

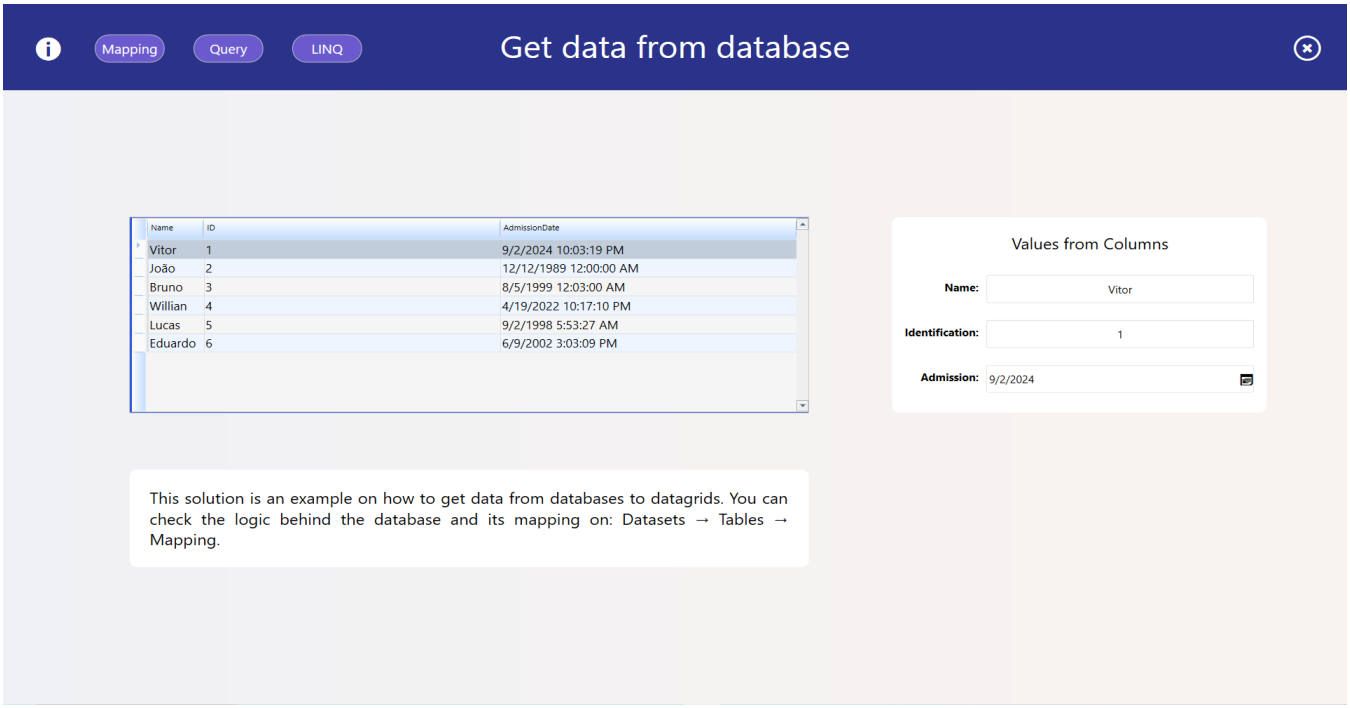
- Using the Mapping feature
- Filtering with SQL queries
- Filtering with LINQ

Technical Information

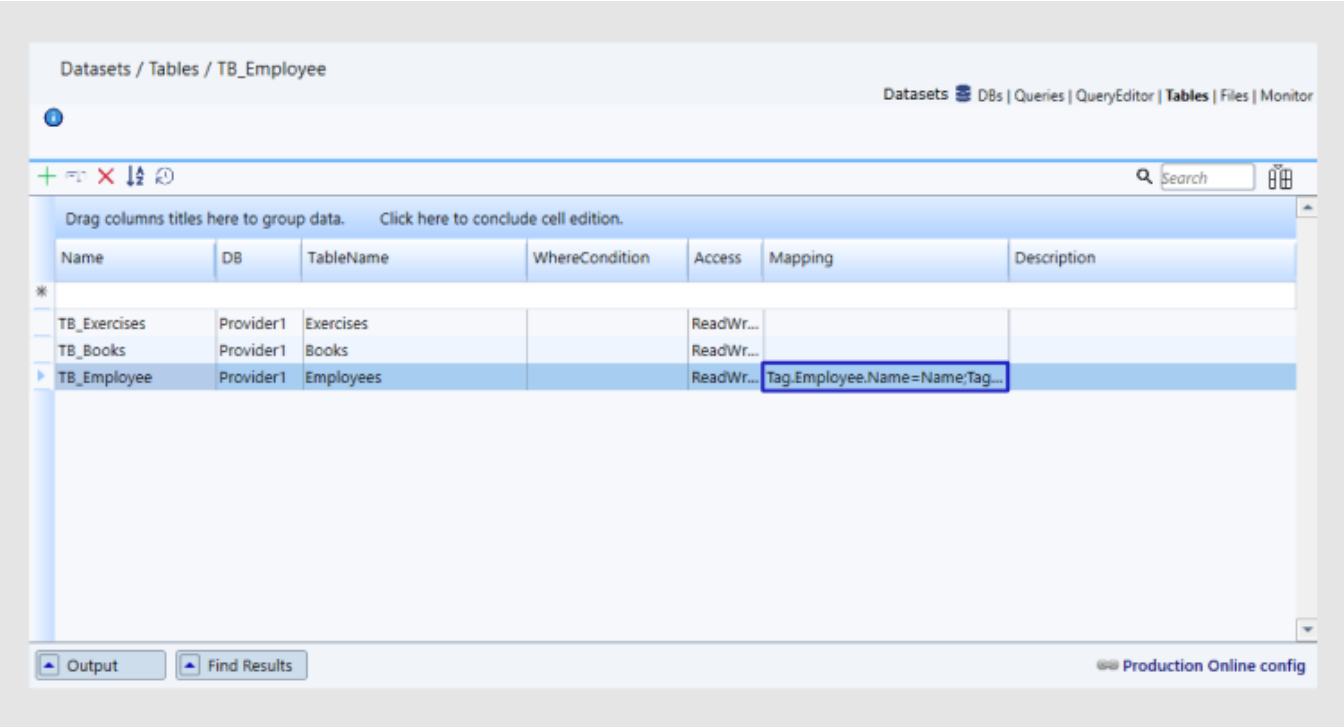
Since this solution explores three different ways to retrieve data from a database, we'll go through each one step by step.

Getting data through Mapping:

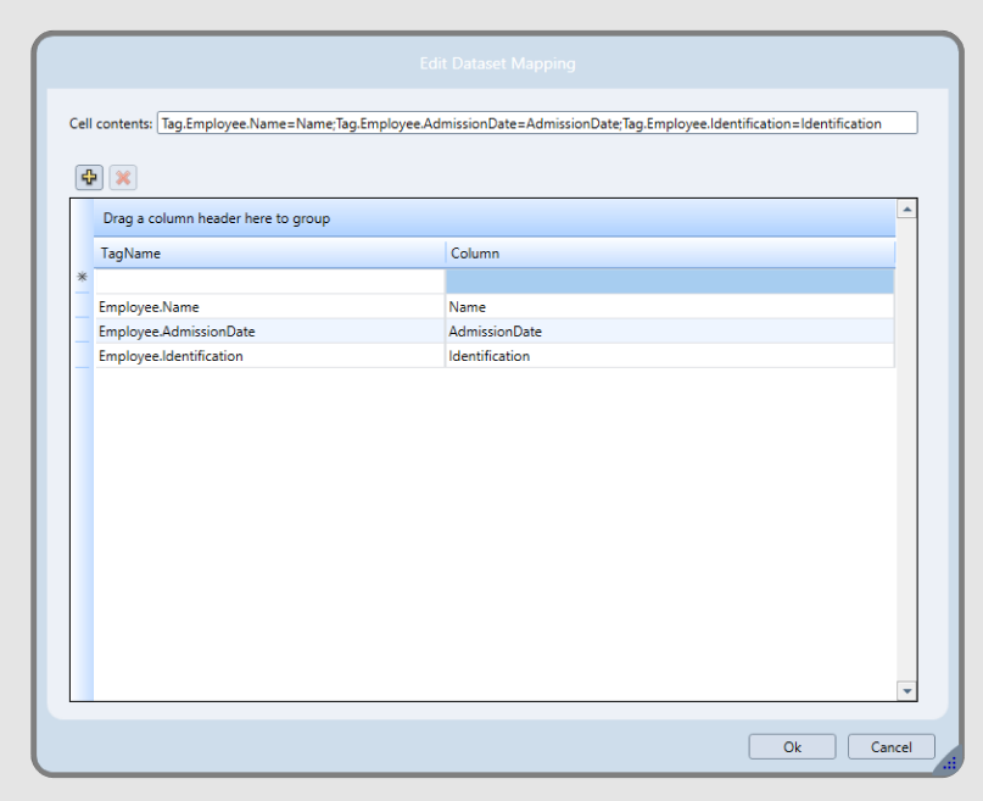
First up, let's take a look at the Mapping feature:



The way this display works is that it pulls the information from the DataGrid when you click on a row, displaying it in the TextBoxes on the right. To make this happen, each column needs to be mapped to a Tag. To do that, go to Dataset/Table and set up your tables there. Then, double-click on the mapping field and click on the "...".

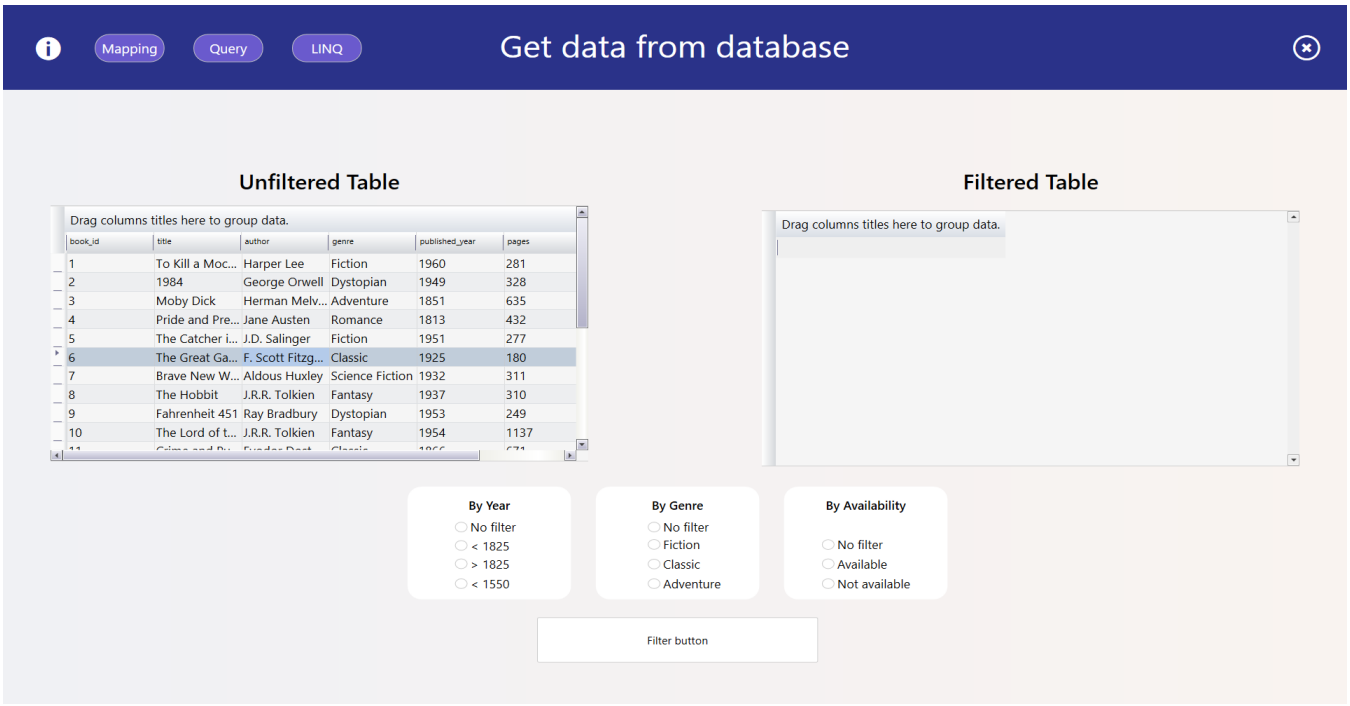


Here you can link each tag to its corresponding column, and it will automatically retrieve the value for the selected row. Alternatively, you can use the InsertCommand method to populate the DataTable with values from those tags.



Filtering table through SQL Queries:

For filtering with SQL Tables, the display works a bit differently since we're now using Radio Buttons for filtering.



This time, each Radio Button plays a role in building a query string by concatenating specific conditions based on the selected option. The method for setting up the string follows this approach:

Build SQL Query

```
string BuildSqlQuery(string yearFilter, string genreFilter, string availabilityFilter) {
        List<string> conditions = new List<string>();

        // Year filter
        if (!string.IsNullOrWhiteSpace(yearFilter) && yearFilter != "nf") {
                if (yearFilter.StartsWith(">") || yearFilter.StartsWith("<") || yearFilter.StartsWith("="))
                        conditions.Add($"published_year {yearFilter}");
        }

        // Genre filter
        if (!string.IsNullOrWhiteSpace(genreFilter) && genreFilter != "nf") {
                conditions.Add($"genre = '{genreFilter}'");
        }

        // Availability filter
        if (!string.IsNullOrWhiteSpace(availabilityFilter) && availabilityFilter != "nf") {
                conditions.Add($"available = {availabilityFilter}");
        }

        // Base query
        string query = "SELECT * FROM Books";

        // Add WHERE clause if there are conditions
        if (conditions.Count > 0) {
                query += " WHERE " + string.Join(" AND ", conditions);
        }

        return query;
}
```

In simpler terms, it checks which filters are selected, adds them to a list of conditions, and then combines them into a single string that gets attached to the WHERE clause in the query.

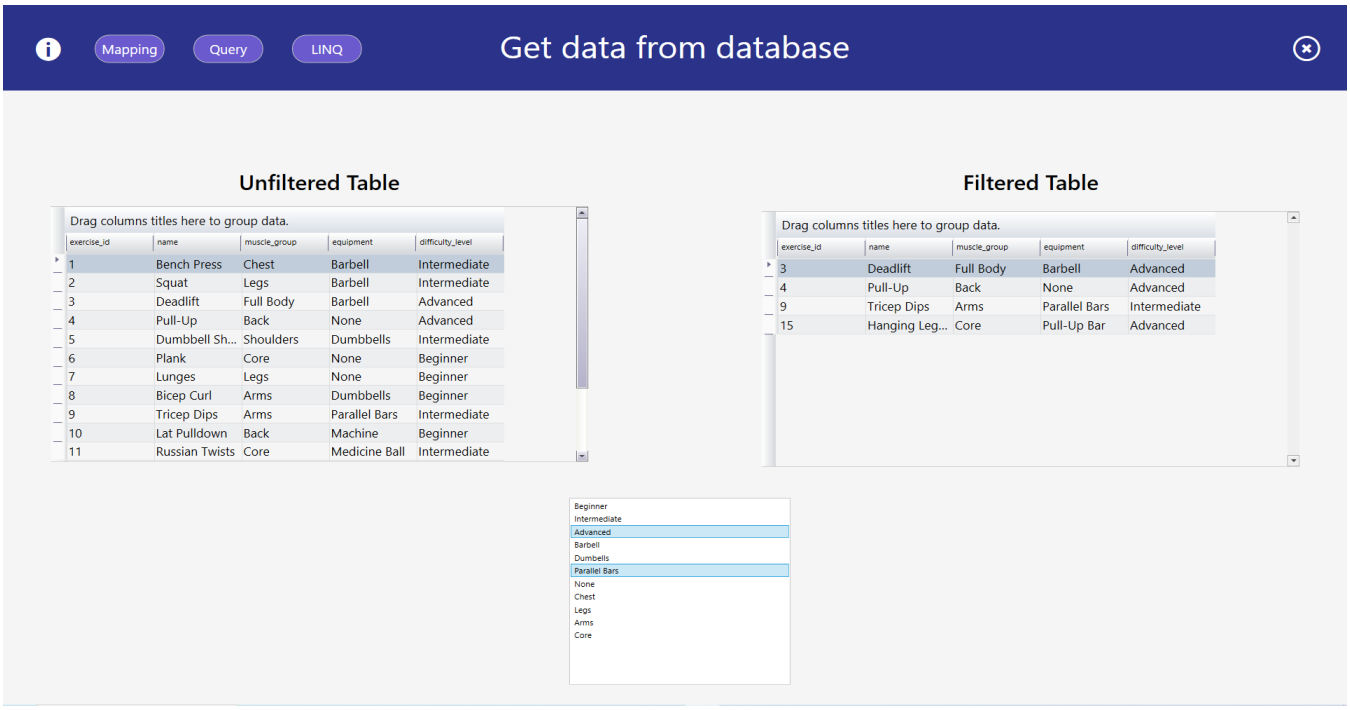
Note: This is a method that needs to be called with some parameters, and it just returns our query. Therefore, you still need to run a SelectCommand, seen below:

MyFilterCall

```
public async Task MyFilterCall(object sender, System.Windows.Input.InputEventArgs e)
{
        @Dataset.Query.Query1.SqlStatement = BuildSqlQuery(@Tag.FilterYear, @Tag.FilterGenre, @Tag.
FilterAvail);
        await @Dataset.Query.Query1.SelectCommandAsync();
}
```

Filtering table through LINQ statements:

Lastly, on our final page, we've added a ListBox with multi-selection enabled to filter the table.



Through CodeBehind, we need to manage the ListBox to configure its settings, retrieve selected values, and set up a subscription to our EventHandler so filtering happens automatically, unlike the previous display, which used a button. For that, we use the following code snippet:

MyListBox

```
TListBox myLB = new TListBox();

public async Task DisplayOpening()
{
        // Retrieves data from datatable
        @Tag.MyDT = await @Dataset.Table.TB_Exercises.SelectCommandAsync();
        @Tag.MyFilteredDT.Value = @Tag.MyDT.Value;

        // Gets control of the ListBox in Drawing
        myLB = this.CurrentDisplay.GetControl("ListBox") as TListBox;
        myLB.SelectionMode = SelectionMode.Multiple;

        // Sets an event handler for SelectionChanged (as in, when a selected item changes state, triggers
the event)
        myLB.SelectionChanged += MyFilterList;
}
```

We also use our DisplayOpening to initialize our DataTables, since we are now using Tags instead of querying the table directly. Also, note that we have set up the Filtered table the same way as the other one, so the full table will be displayed if no filter is applied.

Now, for our EventHandler, we are using a filter that accumulates selections instead of narrowing them down. This means that if you select “chest” and “legs,” you’ll see information for both categories combined, rather than just the entries that contain both. The code snippet for it can be seen below:

MyFilterList

```
public void MyFilterList(object sender, SelectionChangedEventArgs e) {

        // Uses LINQ to properly filter the table
        var rows = @Tag.MyDT.Value.AsEnumerable()
    .Where(a => myLB.SelectedItems.Contains(a.Field<string>("muscle_group")) ||
                myLB.SelectedItems.Contains(a.Field<string>("equipment")) ||
                myLB.SelectedItems.Contains(a.Field<string>("difficulty_level")));

        // If table is empty, show the whole table, otherwise show the filtered table
        if (rows.Any())
                @Tag.MyFilteredDT = rows.CopyToDataTable();
        else
                @Tag.MyFilteredDT = @Tag.MyDT;
}
```

In this section...
