**Developer**

**Survey Configurator App**

Year 2023  
Version 1.00

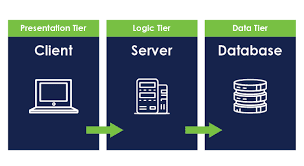
# Description of the App

## Purpose of the app

The Survey Configurator app is a powerful app designed to empower users to create and customize surveys effortlessly. With it is user-friendly interface and flexible options. By providing a wide range of question types, users can gather valuable feedback from respondents, enabling better decision making and enhancing their understanding of various topics.

# Architecture and Design

The application is created utilizing the three-tier architecture and the C# .NET Framework with windows forms. This architecture encourages the separation of concerns and maintainability. Let’s investigate each layer and how it contributes to the application:



## Presentation Layer

1. The top layer is where users connect, and it offers a user-friendly interface for smooth communication.
2. It has forms, controls, and user interface elements for data input and display that were created using windows forms.
3. Handles user input and retrieves data from the **Data Layer** using the **Logic Layer.**

## Logic Layer

1. Work as a bridge between the **Data Layer** and **Presentation Layer.**
2. Contains business logic.
3. Handles data **transformation** and data **validation**.
4. Reduce the link between the presentation and data layer.
5. Improves maintainability.

## Data Layer

1. Handles CRUD (Create, Read, Update, Delete) operations with direct connection with ADO.NET

## Domain Layer

1. Contains crucial classes that serve the app.

## Helper Layer

1. contains assistance functions, and classes that are used at several layers.

# Database Schema

## Tables

### Question

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| Id | Integer | Primary Key | Unique identifier for the question. Used to identify the row in the table |
| Order | Integer | Unique, Not Null | Order of the question in the survey. |
| Text | String | Not Null, Length Between 10 to 1500 | Text of the question. |
| Type Number | Integer | Not Null | Represents the question type as the following:  **1** for **Faces** question  **2** for **Slider** question  **3** for **Stars** question |

### Question Faces

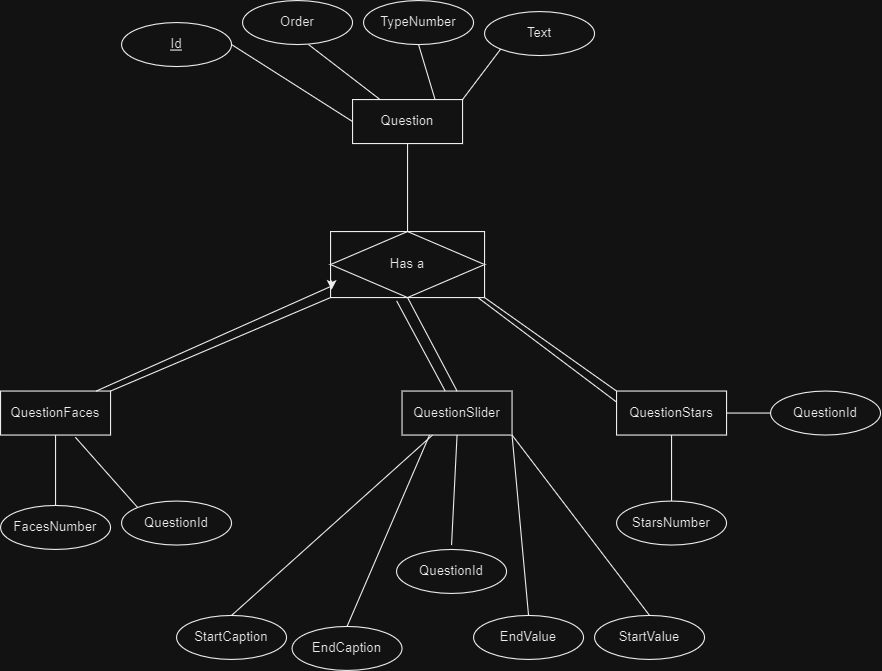
|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| Question Id | Integer | Foreign Key Reference question(Id) | A **unique identifier referencing** the parent question in the **'Question'** table. |
| Faces Number | Integer | Not Null, Range: 2 to 5 | The number of **Faces** that will display in the question |

### Question Stars

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| Question Id | Integer | Foreign Key Reference question(Id) | A **unique identifier referencing** the parent question in the **'Question'** table. |
| Stars Number | Integer | Not Null, Range: 1 to 10 | The number of **Stars** that will display in the question |

### Question Slider

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| Question Id | Integer | Foreign Key Reference question(Id) | A **unique identifier referencing** the parent question in the **'Question'** table. |
| Start Value | Integer | Not Null, Range:>=0 and < EndValue | **Start** value for the slider. |
| End Value | Integer | Not Null, Range > StartValue and <=100 | **End** value for the slider. |
| Start Caption | String | Not Null, Length: 3-500 characters | **Caption** for the **Start** value. |
| End Caption | String | Not Null, Length: 3-500 characters | **Caption** for the **End** value. |



# Code Overview

To improve readability, maintainability, and consistency in the codebase, specific rules and patterns have been followed. Here is a summary overview of the important aspects and conventions used throughout the code:

## Names of Function and Parameters

1. All function names start with a capital letter.
2. To indicate function parameters, using the letter “p” in front of the parameter name. The developer can distinguish parameters within the function.

Example:

public int GetQuestions(ref List<Question> **pList**)

{

try

{

return dbQuestion.GetQuestions(ref pList);

}

catch (Exception e)

{

Log.Error(e);

return ResultCode.ERROR;

}

}

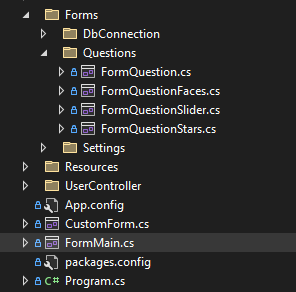
1. Temporary Variables are named with letter “t” added in the front. For example, “**tResult**”.

## Try-Catch

1. Try-Catch blocks are used throughout the app.
2. This strategy guarantees error handling and runtime graceful recovery from exceptions.
3. The application maintains stability and provides helpful error signals for debugging.

## Presentation Layer

1. A custom form has been created for inheritance by other forms, simplified changes are made to all forms at once.
2. The FormMain acts as the primary user interface for the users to interact with the survey and display all questions.
3. The forms folder has several subfolders, including **Settings** and **Questions.**

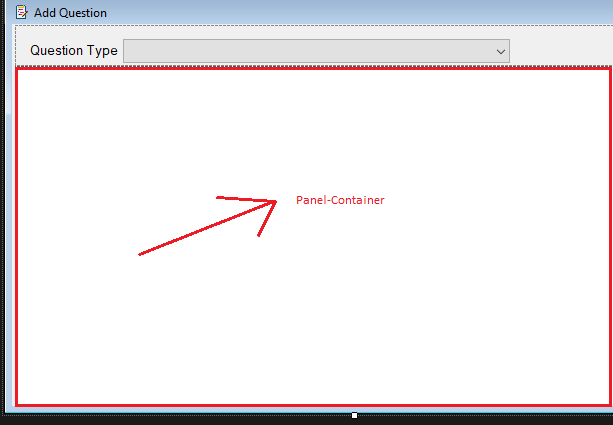


### Question Forms

1. Each question type has a specific form within the “Questions” folder.
2. In order to display the proper form based on the question type and provide dynamic UI elements for various question types, FormQuestion is used.

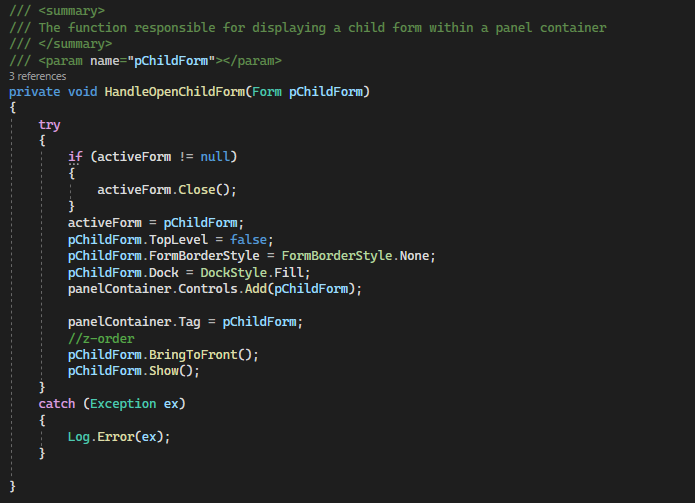
#### Dynamic Form Question

Since each question has its own form and can be run separately, we use the form and run it inside a panel in the "FormQuestion" as follows.



inside the FormQuestion we created the HandleOpenChildForm function, taking a form as a parameter and opening it inside the panel.

Example:



This is our approach to managing the user interface when the question type is changed.

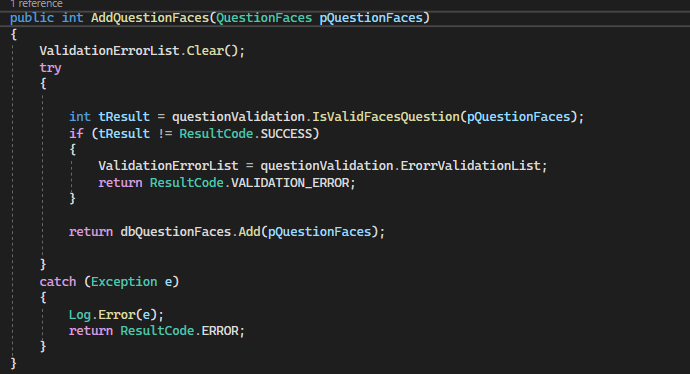
## Code Flow

### Add Question

Let’s take **Question Faces** as an example

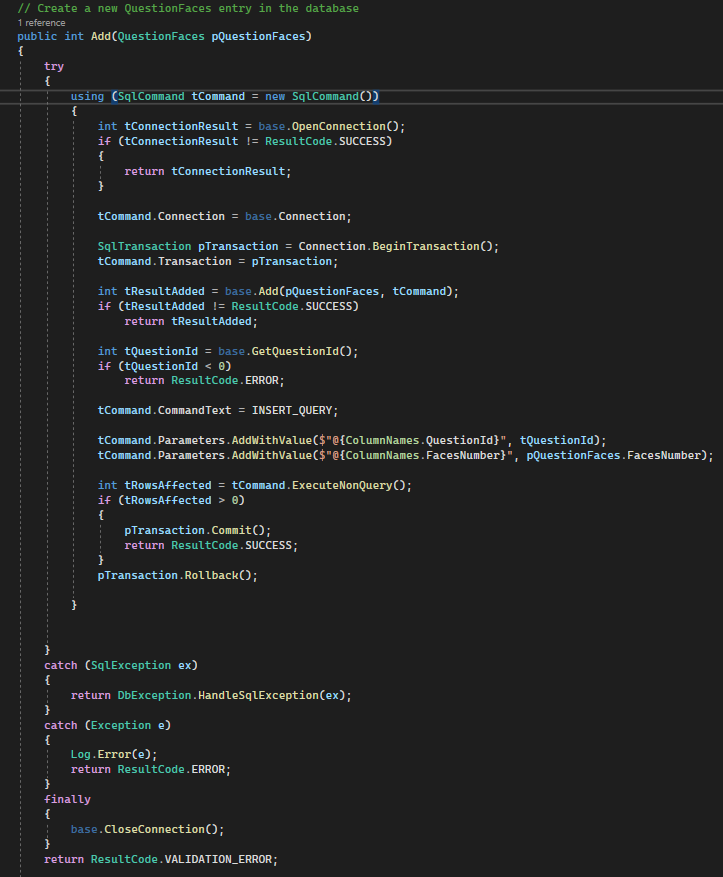
1. **User Interface**
   * The user interacts with the UI and enters the required info to create a new question of type Faces.
   * Once the “Save” button has been clicked, the data will transfer to the Question Manager class.
   * Here is an example that show you how we call the Add function: tResult = questoinManager.AddQuestionFaces(questionFaces);
2. **Logic Layer Question Manager**
   * The QuestionManager class receives the user's inputs and invokes the AddQuestionFaces function, passing the QuestionFaces object as a parameter.
   * The input data check for to preliminary validation by the question manager to ensure it meets the requirements for the “**Faces**” type.

Example:



1. **Input Validation**
   * The QuestionFaces object is validated by the AddQuestionFaces function.
   * If any input fails validation (e.g., missing fields, invalid range for Faces Number), the function generates a list of errors to communicate the validation issues. The UI can then display the error messages to the user, prompting them to correct the input accordingly.
   * If the input data successfully passes validation, the QuestionManager moves to the next step.
2. **Data Layer**
   * The QuestionManager then gives the **DbQuestionFaces** class in the Data Layer the QuestionFaces object that has been verified.
   * The DbQuestionFaces class handles the data storage process using ADO.NET.
   * If the data insertion is successful, the "**Faces**" question is stored in the database successfully.
   * However, if an error occurs during the database interaction (e.g., connection issues, data conflicts), the **DbQuestionFaces** will return a status code representing the specific error.

Example:



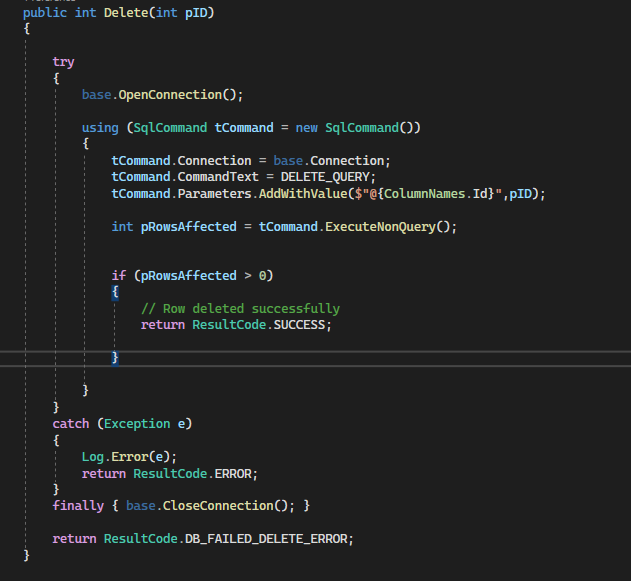
**The Same thing for the other types of questions**

### Delete Question

1. User Interface (UI):

* The user selects a question in the UI by clicking on it.
* The user then clicks the "Delete" button, indicating their intention to remove the selected question.

1. Logic Layer QuestionManager
   * The QuestionManager class receives the selected question ID as a parameter.
   * It then calls the Delete method in the Data Layer's DBQuestion class, passing the question ID as a parameter.
2. Data Layer DbQuestion delete method will handle the delete operation



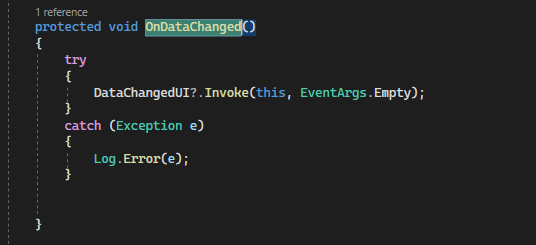
### Refresh UI

By establishing a background thread in the logic layer to recheck for data changes to handle the list view in the presentation layer.

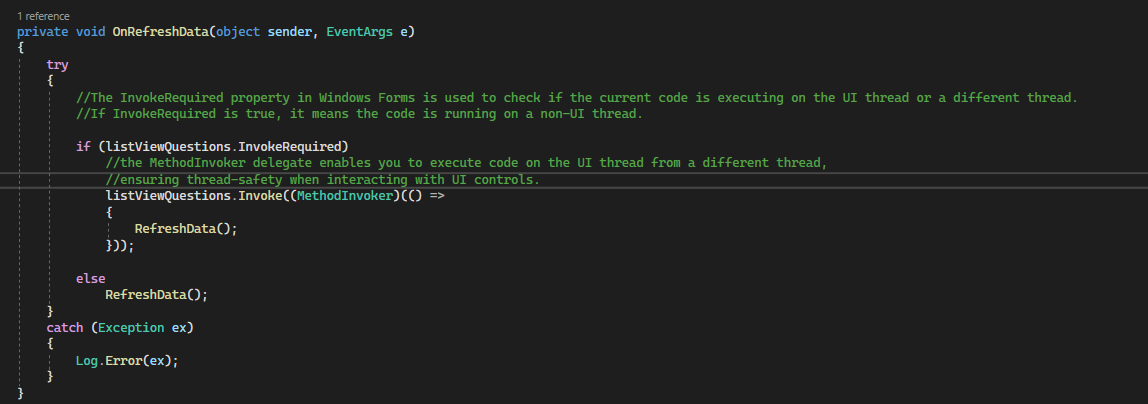
1. **Logic Layer**
   1. **FollowDbChanges** Method
      * We created a background thread using **ThreadStart** that repeatedly checks for data changes after a specific duration (REFRESH\_DURATION).
      * Inside the thread, we call **GetQuestions()** to retrieve the latest data from the Data Layer and store it in a local list variable (**questionsList**).
      * We empty the **questionsList**, update it with the most recent data, and call the **OnDataChanged**() event to notify the Presentation Layer if the data is not equal or if it is the first call.



* 1. **OnDataChanged** Method
     + The DataChangedUI event, which notifies the UI that the data has changed, is raised by the OnDataChanged() function.
     + We use the DataChangedUI?.Invoke(this, EventArgs.Empty); syntax to invoke the event.

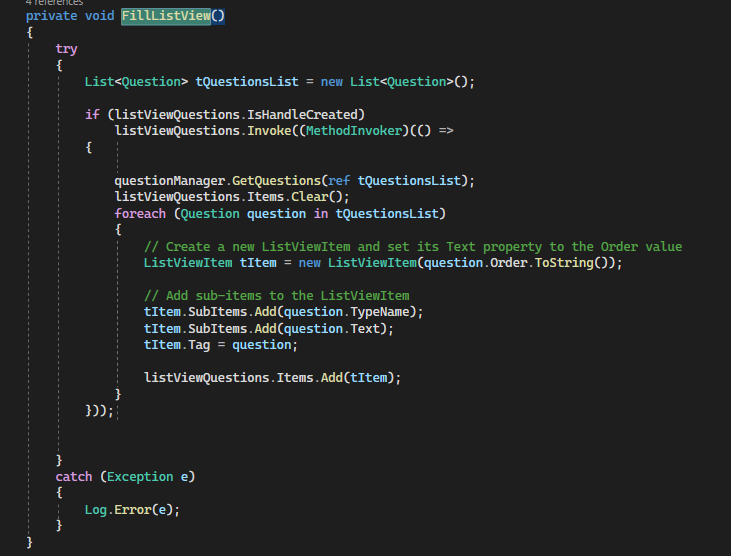


1. **Presentation Layer**
   1. We use questionManager in the Presentation Layer to listen the **DataChangedUI** event.
   2. The **OnRefreshData**() method is triggered whenever the **DataChangedUI** event is raised.
      * Inside **OnRefreshData**(), we check if the UI control (**listViewQuestions**) requires invoking due to being accessed from a different thread.
      * The **RefreshData()** method is then called, which triggers the list view data refresh.



* 1. **RefreshData**() is responsible for refreshing the data in the list view (**listViewQuestions**).
     + To fill the list view with the updated data we use **FillListView** function





# Error Handling and Logging

To improve program reliability and ease debugging, we implemented an effective error handling and logging mechanism in the **Helper** layer of our application:

## Log Class

1. The Log class provides a robust logging system that may store variety of data types in the log file.
2. It allows us a way to monitor and analyze important data, including exceptions, informative messages, and warnings.
3. The log class allows developers to record critical actions.

## Status Code Class

1. The application's status codes are represented by constant variables in the StatusCode class.
2. These status codes help identify and communication of particular results and faults.
3. We guarantee ease of maintenance across the entire application.

## Benefits of Error Handling and Logging

1. Troubleshooting
2. Monitoring and Analytics
3. Fixing Errors

# Conclusion

This developer's guide gives our development team the tools they need to efficiently create, improve, and manage the program. Developers are empowered to build features, follow coding standards, and take use of the Domain and Helper layers for more efficient development when they have a comprehensive understanding of the three-tier design.